

**IMPLEMENTATION OF NOISE CANCELLATION
WITH
HARDWARE DESCRIPTION LANGUAGE**

By

LEE KUANG SUN

FINAL PROJECT REPORT

Submitted to the Electrical & Electronics Engineering Programme
in Partial Fulfillment of the Requirements
for the Degree
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

Universiti Teknologi Petronas
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

© Copyright 2006
by
Lee Kuang Sun

CERTIFICATION OF APPROVAL

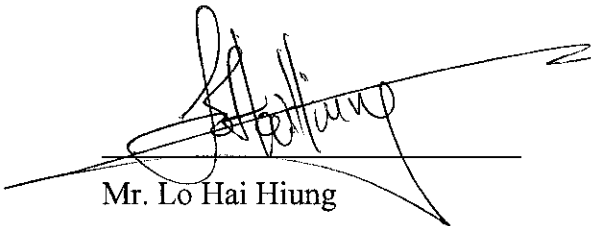
IMPLEMENTATION OF NOISE CANCELLATION WITH HARDWARE DESCRIPTION LANGUAGE

by

Lee Kuang Sun

A project dissertation submitted to the
Electrical & Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfilment of the requirement for the
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

Approved:

A handwritten signature in black ink, appearing to read 'Lo Hai Hiung', is written over a horizontal line. The signature is stylized and extends to the right.

Mr. Lo Hai Hiung
Project Supervisor

UNIVERSITI TEKNOLOGI PETRONAS
TRONOH, PERAK

June 2006

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.

A handwritten signature in black ink, appearing to read 'Lee Kuang Sun', is written over a horizontal line.

Lee Kuang Sun

ABSTRACT

The objective of this project is to implement noise cancellation technique on an FPGA using Hardware Description Language. The performance of several adaptive algorithms is compared to determine the desirable algorithm used for adaptive noise cancellation system. The project will focus on the implementation of adaptive filter with least-mean-squares (LMS) algorithm or normalized least-mean-squares (NLMS) algorithm to cancel acoustic noises. This noise consists of extraneous or unwanted waveforms that can interfere with communication. Due to the simplicity and effectiveness of adaptive noise cancellation technique, it is used to remove the noise component from the desired signal. The project is divided into four main parts: research, Matlab simulation, ModelSim simulation and hardware implementation. The project starts with research on several noise cancellation techniques, and then with Matlab code, Simulink and FDA tool, the adaptive noise cancellation system is designed with the implementation of the LMS algorithm, NLMS algorithm and recursive-least-square algorithm to remove the interference noise. By using the Matlab code and Simulink, the noise that interfered with a sinusoidal signal and a record of music can be removed. The original signal in turns can be retrieved from the noise corrupted signal by changing the coefficient of the filter. Since filter is the important component in adaptive filtering process, the filter is designed first before adding adaptive algorithm. A Finite Impulse Response (FIR) filter is designed and the desired result of functional simulation and timing simulation is obtained through ModelSim and Integrated Software Environment (ISE) software and FPGA implementation. Finally the adaptive algorithm is added to the filter, and implemented in the FPGA. The noise is greatly reduced in Matlab simulation, functional simulation and timing simulation. Hence the results of this project show that noise cancellation with adaptive filter is feasible.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincere gratitude and appreciation to several parties who have given me guidance, support and invaluable advice. Most importantly, I would like to express my deepest gratitude to my family, especially my mum and my dad. Their love, care and support have always been an encouragement for me to move forward.

Sincere gratitude also goes to my supervisor Mr. Lo Hai Hiung for allocate his time to supervise and guide me throughout the project. It has been a pleasure working under the direction of my supervisor, which introduce me to the advanced hardware design for adaptive filter. Thanks for his patience, inspiration, contribution of precious ideas and constant guidance. I would like to thanks Dr. Yap Vooi Voon and Mr. Patrick Sebastian for their counsel, support and encouragement to complete my final year project.

Lastly, I would like to thank the lecturers and staffs of the Electrical and Electronic Engineering Department of Universiti Teknologi PETRONAS for their support and assistance in completing my final year project. Despite of their many responsibilities, they have always cheerfully to accommodate our requests, especially Ms. Azirawati Aziz and Ms. Siti Hawa who had giving untiring commitment to the project. Thanks to those who had directly or indirectly help me in this project. Their willingness to cooperate and assist had helped me to complete the project.

TABLE OF CONTENTS

LIST OF FIGURES	x
LIST OF TABLES	xi
LIST OF ABBREVIATIONS	xii
CHAPTER 1: INTRODUCTION	1
1.1 Background study	1
1.2 Problem statement	2
1.3 Objective	2
1.4 Scope of study	3
1.5 Organization of report	3
CHAPTER 2: LITERATURE REVIEW AND/OR THEORY	4
2.1 Active noise control	4
2.2 Adaptive filtering	4
2.3 Principle of orthogonality	5
2.4 Adaptive algorithm	8
2.5 Least-Mean-Square algorithm	8
2.6 Normalized Least-Mean-Square algorithm	10
2.7 Recursive-Least-Squares algorithm	10
2.8 Feasibility of fixed-point transversal adaptive filter in FPGA	11
2.9 Comparison of DSP processor with FPGA	12
2.10 Other noise cancellation techniques	13
2.11 HDL	13
2.12 Verilog	14
2.13 FPGA	14
2.14 Multiplier	14
2.15 Multiplication of signed binary number	16
2.16 Multiplication of fractions number	17

CHAPTER 3: METHODOLOGY/ PROJECT WORK	18
3.1 Research	18
3.2 Matlab simulation	19
3.3 ModelSim simulation	21
3.4 Hardware implementation	26
3.4.1 Tools required	26
3.4.2 Design flow	27
3.4.3 Design entry	28
3.4.4 Design simulation	28
3.4.5 Creating and Editing Timing and Area Constraints	29
3.4.6 Design Synthesis and Implementation	29
3.4.7 Verification of the Implemented Design	30
3.4.8 Creating Configuration Data	31
 CHAPTER 4: RESULT AND DISCUSSION	 32
4.1 Matlab simulation	32
4.2 ModelSim simulation	35
4.2.1 FIR filter	35
4.2.2 Adaptive filter	38
4.3 Hardware implementation	40
 CHAPTER 5: CONCLUSION AND RECOMMENDATIONS	 42
5.1 Conclusion	42
5.2 Recommendations	44
 REFERENCE	 46

APPENDICES		48
APPENDIX A	Gantt chart of final year project activities	48
APPENDIX B	Matlab code for sound recording and noise generation	49
APPENDIX C	Matlab code of noise cancellation with adaptive filter	50
APPENDIX D	Verilog code for a 10 order direct-form FIR filter	51
APPENDIX E	Verilog code for a 10 order adaptive filter	56
APPENDIX F	Performance of Virtex-II	63
APPENDIX G	Simulation result of adaptive noise cancellation system with simulink	65

LIST OF FIGURES

Figure 2.1 Block diagram for the adaptive filtering	5
Figure 2.2 Steps involved in computing the linear discrete-time form of convolution ...	6
Figure 2.3 Adaptive filtering on noise cancellation application	8
Figure 2.4 Block diagram of binary multiplier	15
Figure 2.5 Representation of decimal value -0.02130126953125 in fractional number	17
Figure 3.1 Methodology of adaptive filter design	18
Figure 3.2 Simulink for waveform observation with FIR filter and LMS algorithm	20
Figure 3.3 Simulink for sound observation with FIR filter and LMS algorithm	20
Figure 3.4 Filter design with FDA tool.....	21
Figure 3.5 Ten orders FIR filter structure	23
Figure 3.6 Flow chart of filter design in Verilog	24
Figure 3.7 Extended flow chart of adaptive filter design.....	25
Figure 3.8 Virtex-II Development Kit	26
Figure 3.9 General FPGA Design Stages	27
Figure 3.10 FPGA Editor - Detailed view of filter design for an output pin.....	31
Figure 4.1 Adaptive filter response.....	34
Figure 4.2 The original signal, noise, desired signal and the error signal	35
Figure 4.3 Result of functional simulation with ModelSim	36
Figure 4.4 Input and output of the filter in Matlab	36
Figure 4.5 The coefficients of filter in floating-point precision	37
Figure 4.6 The coefficients of filter in fixed-point precision.....	38
Figure 4.7 Output of filter with zero error	38
Figure 4.8 Output of adaptive filter with error	39
Figure 4.9 Verilog coding for implementation of equation (5)	40
Figure 4.10 Timing simulation result for FIR filter.....	41
Figure 4.11 Timing simulation result for adaptive filter.....	41

LIST OF TABLES

Table 2.1 Description of variables in LMS algorithm equations..... 9

Table 2.2 Description of variables in RLS algorithm equations..... 11

Table 2.3 Performance comparison between DSP processor and FPGA 12

Table 4.1 Summary of the observation from the adaptive noise cancellation system... 33

LIST OF ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
DDR	Double Data Rate
DSP	Digital Signal Processing
EDIF	Electronic Data Interchange File
FIR	Finite Impulse Responed
FPGA	Field Programmable Gate Array
GMACs	Gillion of Multiply Accumulates per second
HDL	Hardware Description Language
ICA	Independent Component Analysis
IIR	Infinite Impulse Response
IOB	Input/output block
ISE	Integrated Software Environment
JTAG	Joint Test Action Group
LE	Logic Element
LMS	Leas-Mean-Square
LUT	Look-up Table
LVDS	Low Voltage Differential Signaling
MIPS	Million Instructions per Second
NCF	Netlist Constraint File
PROM	Programmable Read Only Memory
RBF	Radial Basis Function
RLS	Recursive-Least-Squares
RTL	Register Transfer Level
UCF	User Constraints File
VLSI	Very Large Scale Integration
XST	Xilinx Synthesis Technology

CHAPTER 1

INTRODUCTION

1.1 Background Study

Noise means unwanted sound commonly interferes with normal hearing. But in electronics, noise can refer to the electronic signal corresponding to acoustic noise in an audio system or the electronic signal corresponding to the noise commonly seen as 'snow' on a degraded television or video image. In signal processing or computing it can be considered data without meaning; that is, data that is not being used to transmit a signal, but is simply produced as an unwanted by-product of other activities.

Noise cancellation is a method for reducing or cancelling out undesirable sound. It is often called Active Noise Cancellation because the electronics involved actively cause the noise reduction in real time. One of the popular methods used in noise cancellation is adaptive noise cancellation. Adaptive noise canceling is an approach to reduce noise based on reference noise signals. It is used in communication systems that are contained on a single silicon chip, where real-time processing is required.

This adaptive noise cancellation can be implemented in the Field Programmable Gate Array (FPGA) with Hardware Description Language (HDL). HDL is a textual description of a system or circuitry. It allows the expression of the concepts that previously could not be expressed by manual notations, such as Algorithm State Machine (ASM) notation and circuit diagram [1].

1.2 Problem Statement

Noise consists of extraneous or unwanted waveforms that can interfere with communication. It is most often associated with irritating unwanted signals. Another form of noise is electronic noise that can interfere with electromagnetic communication. There are methods to suppress or reduce the noise. However the conventional method such as wearing special earphone with simple filtering process is not as effective as adaptive noise cancellation, which uses the adaptive filter with least-mean-squares (LMS) algorithm to remove the noise component from the desired signal [2].

Moreover the implementation of this noise cancellation can be done by using the Hardware Description Language (Verilog or VHDL) to program into an FPGA as demonstrated in [5]. The FPGA maintains the high specificity of the Application Specific Integrated Circuit (ASIC) while avoiding its high development cost and its inability to accommodate design modifications after production. Highly adaptable and design-flexible, FPGAs provide optimal device utilization through conservation of board space and system power-important advantages not available with many stand-alone Digital Signal Processing (DSP) chips [22].

1.3 Objective

The objective of this project is to implement adaptive noise cancellation in FPGA with the Hardware Description Language (Verilog). The performance of several algorithms used in adaptive filter is compared to determine the suitable algorithm used for real time application. These algorithms include least-mean-squares (LMS), normalized least-mean-squares (NLMS) and recursive-least-squares (RLS) [3]. The project may focus on the implementation of adaptive filter with least-mean-squares (LMS) algorithm to cancel noise, since it is theoretically less involved in mathematics calculation. Besides that, the performance of finite impulse response filter (FIR) and infinite impulse response filter (IIR) in implementation of adaptive filter is compared to determine the suitable filter, which can work well with the algorithms of adaptive filter.

1.4 Scope of Study

This noise cancellation technique can be used in hearing aids, telephones and other communication devices. The algorithm of adaptive filter is commonly implemented to cancel the noise of engine in the aeroplane or ship. In addition, the adaptive filter is also utilized in image processing to produce a clearer image.

This project is started with the Matlab simulation to implement cancellation of noise in a single sinusoidal signal, and cancellation of noise that is interfering with a song or recorded sound. From the result of Matlab simulation, the suitable algorithm and filter is selected to implement the adaptive filter used for the noise cancellation.

The adaptive filter design in the Matlab is developed with the Verilog language. This covers the design of basic FIR filter with limited number of input and output pins. Then the algorithm of adaptive filter is implemented to the FIR filter, which continuously updates the weight (coefficient) of the filter. The design is tested with the functional simulation and timing simulation before implemented on an FPGA.

1.5 Organization of Report

The report begins with the overview of the noise, noise cancellation techniques, Hardware Description Language and project objectives and scope of study as mentioned above. Next, the reader is introduced to the noise cancellation techniques, algorithms of adaptive filter, Verilog, FPGA performance and method of multiplication. The report is followed by two main themes, the methodology section, and result and discussion section. The methodology covers research, Matlab simulation, ModelSim simulation and hardware implementation. The results of simulations are discussed subsequently. Finally, several recommendations for adaptive filter implementation are given after the conclusion.

CHAPTER 2

LITERATURE REVIEW AND THEORY

This chapter includes several noise cancellation techniques, adaptive filtering, adaptive algorithm, and the comparison between DSP processor with FPGA.

2.1 Active Noise Control

Active noise control (ANC) destructs interference of propagating acoustic waves. The acoustic wave interference can be controlled to produce zone of quietness by using the DSP devices to design and implement digital ANC systems that operate in real-time. The most practical ANC systems are using adaptive filtering techniques, which allow the system to adaptively model the acoustic paths [2].

The inherent filter inside the active noise controller can either be a finite impulse response filter (FIR) or an infinite impulse response filter (IIR). There are advantages and disadvantages for each type of filter in this application. FIR filter are stable and the filter coefficients are easier to handle compared with the IIR filters as FIR filter uses the forward paths only. But the order of FIR filter required is much higher compared to the IIR filter with similar spectra characteristics. On the other hand, an IIR filter involves both feedforward and feedback paths. The presence of the feedback means that portion of the filter output and possibly other internal variables in the filter are fed back to the input. This will cause it to be unstable if the filter is not designed properly [3].

2.2 Adaptive Filtering

The goal of any filter is to extract useful information from noisy data. Whereas a normal filter is designed in advance with knowledge of the statistics of both the signal and the unwanted noise, the adaptive filter continuously adjusts to a changing environment through the use of least-mean-squares (LMS) algorithm,

normalized-least-mean-squares (NLMS) algorithm or recursive recursive-least-squares (RLS) algorithm. The filter weights are usually adapted or updated using these algorithms. This type of algorithm basically attempts to minimize the mean of the error signal squared. This is useful when the statistics of the signals are not known beforehand.

According to S. Hakyin [3], adaptive filter design can be optimized by minimizing a cost function by using Mean-square value of the estimation error. In particular, the mean-square-error criterion results in second-order cost function dependence on the unknown coefficients in the impulse response of the filter. Moreover, the cost function has a distinct minimum that uniquely defines the optimum statistical design of the filter.

The essence of the filtering problem is summarised with the following statement: Design a linear discrete-time filter whose output, $y(n)$, provides an estimate of a desired response, $d(n)$, given a set of input samples, $u(0), u(1), \dots$, such that the mean-square value of the estimation error, $e(n)$, defined as the difference between the desired response, $d(n)$, and the actual response, $y(n)$, is minimized [3].

The mathematical solution is developed to this statistical optimization problem by following the principle of orthogonality.

2.3 Principle of Orthogonality

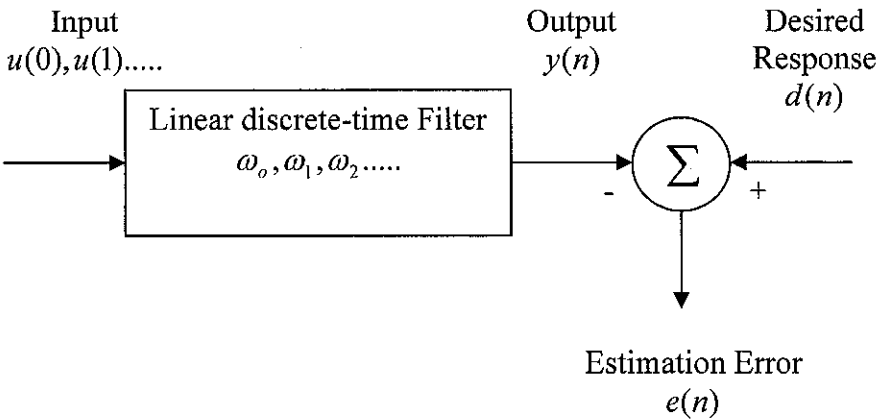


Figure 2.1: Block diagram for the adaptive filtering [3].

As shown in figure 2.1 from S. Hakyin [3], the filter input is denoted by the time series $u(0), u(1), u(2), \dots$, and the impulse response of the filter is denoted by w_0, w_1, w_2, \dots , both of which are assumed to have complex values and infinite duration. The filter output at a discrete time n is defined by the linear convolution sum

$$y(n) = \sum_{k=0}^{\infty} w_k^* u(n-k), \quad n = 0, 1, 2, \dots \quad (1)$$

where the asterisk denotes complex conjugation. Note that, in complex terminology, the term $w_k^* u(n-k)$ represents the scalar version of an inner product of the filter coefficient w_k and the filter input $u(n-k)$. Figure 2.2 illustrates the steps involved in computing the linear discrete-time form of convolution described in equation (1) for real data.

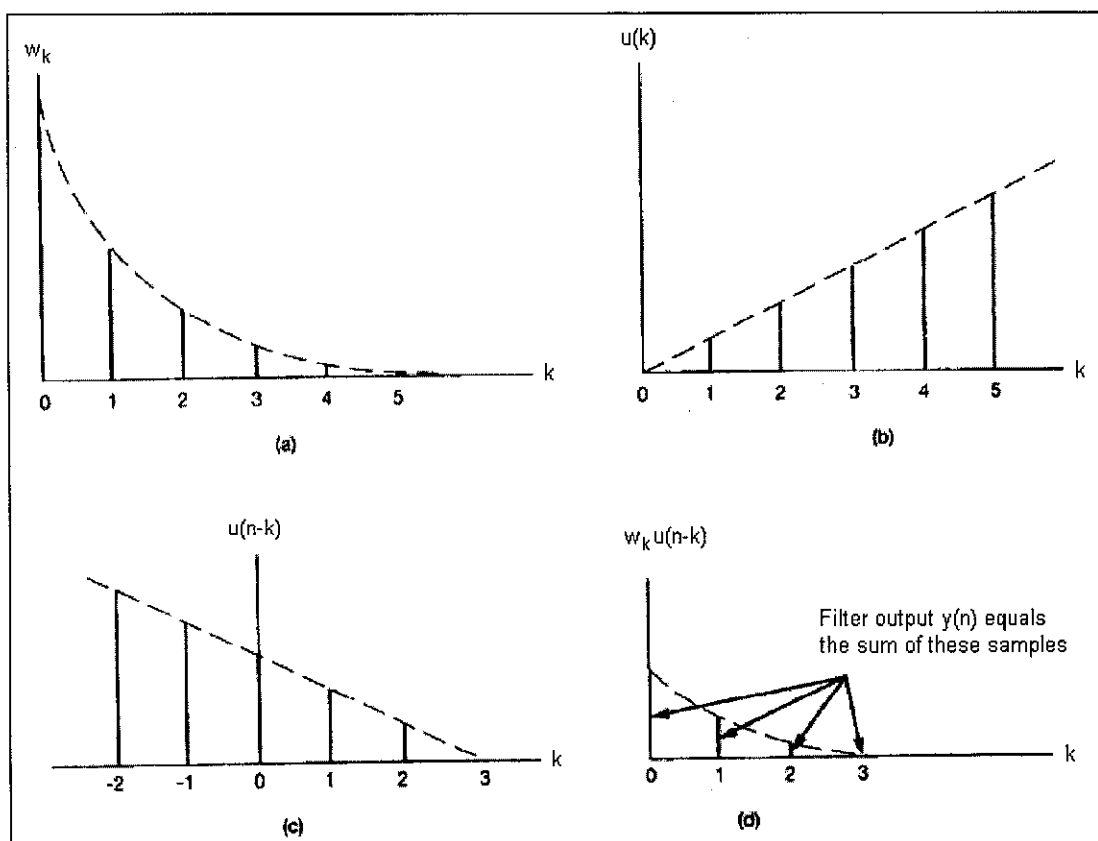


Figure 2.2: Steps involved in computing the linear discrete-time form of convolution

(Source: S. Hakyin, 2002. Adaptive Filter Theory. Prentice Hall, Inc.)

From S. Hakyin [3], the purpose of the filter in figure 2.1 is to produce an estimate of the desired response, $d(n)$. It is assumed that the filter input and the desired response are single realizations of jointly wide-sense stationary stochastic processes, both with zero mean. If the means are nonzero, simply subtract them from the input, $u(n)$, and the desired response, $d(n)$, before filtering. The estimation of $d(n)$ is naturally accompanied by an error, which is defined by the difference

$$e(n) = d(n) - y(n) \quad (2)$$

The estimation error, $e(n)$, is the sample value of a random variable. To optimize the filter design, the mean-square value of $e(n)$ should be minimized. Thus the cost function as the mean-square error

$$J = E [e(n) e^*(n)] \quad (3a)$$

$$J = E [|e(n)|^2] \quad (3b)$$

where E denotes the statistical expectation operator. The requirement is therefore to determine the operating conditions under which J attains its minimum value. This error signal is used to incrementally adjust the filter's weights for the next time instant.

Several algorithms exist for the weight adjustment, such as the *Least-Mean-Square* (LMS) and the *Recursive-Least-Squares* (RLS) algorithms. The choice of training algorithm is dependent upon needed convergence time and the computational complexity, as statistics of the operating environment.

There are four basic classes of applications for adaptive filters, which include *identification*, *inverse modeling*, *prediction*, and *interference cancellation*. Figure 2.3 shows one of the adaptive filtering applications, which is noise cancellation.

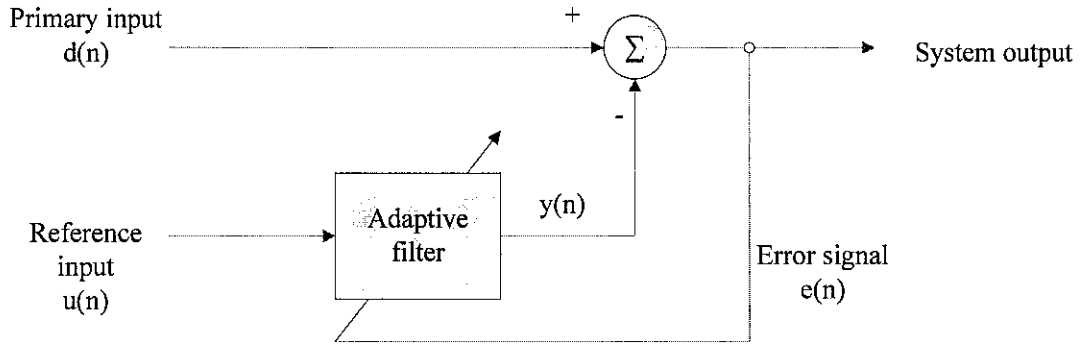


Figure 2.3: Adaptive filtering on noise cancellation application.
(Source: S. Hakyin, 2002. Adaptive Filter Theory. Prentice Hall, Inc.)

An example of adaptive noise cancellation from Matlab [4] is removing the engine noise of airplane from the pilot voice signal. It is clearly shown that adaptive noise cancellation generally does a better job than a classical filtering because the noise is subtracted rather than filtered.

2.4 Adaptive Algorithms

There are numerous methods for performing weight update of an adaptive filter. These include Wiener filter, method of steepest descent, least-mean-square algorithm, recursive-least-squares algorithm and the Kalman filter [3]. The operating environment, signals of interest, convergence time and computation power are the factors considered in deciding the use of algorithm. However for this project, the performance of adaptive algorithms used in adaptive filter is compared before implemented in FPGA, this is to determine the suitable algorithm used for real time application.

2.5 Least-Mean-Square Algorithm

The least-mean-square (LMS) algorithm is similar to the method of steepest-descent in that it adapts the weights by iteratively approaching the Minimum Square Error (MSE). Widrow and Hoff invented this technique in 1960 to train neural networks. The key is that instead of calculating the gradient at every time step, the LMS algorithm uses a rough approximation to the gradient.

The algorithm used to estimates the filter weights, or coefficients, minimize the error, $e(n)$, between the output signal, $y(n)$ and the desired signal, $d(n)$. The algorithm is defined by [3] for the equations (4), (5) and (6).

$$e(n) = d(n) - w^T(n)u(n) \quad (4)$$

$$w(n+1) = w(n) + \mu u(n)e^*(n) \quad (5)$$

The description of the variables is shown in table below

Variable	Description
$e(n)$	The estimation error at time n
$d(n)$	Desired response at time n
$u(n)$	M-by-1 tap-input vector at time n, $[u(n), u(n-1), \dots, u(n-M+1)]^T$
$w(n)$	Tap-weight vector at time n
$w(n+1)$	Estimate of tap-weight vector at time n+1
μ	The adaptation step size

Table 2.1: Description of variables in LMS algorithm equations.

As with the steepest-descent algorithm, it can be shown to converge for values of μ less than the reciprocal of λ_{\max} , but λ_{\max} may be time-varying, and to avoid computing it another criterion can be used [3]. This is

$$0 < \mu < \frac{2}{MS_{\max}} \quad (6)$$

where M is the number of filter taps and S_{\max} is the maximum value of the power spectral density of the tap inputs u.

The LMS algorithm is most widely implemented in practice due to its good performance in real time applications. The number of operations involved for an N-tap filter only are $2*N$ multiplications and N additions per coefficient update [3].

2.6 Normalized Least-Mean-Square (NLMS) Algorithm

As mentioned in the LMS algorithm, the tap-weight of the filter is adjusted as shown in equation (4) and (5). However the adjustment is proportional to the tap-input vector, $u(n)$. As stated by S. Hakyin [3], the LMS filter will suffers from a gradient noise amplification problem when $u(n)$ is large. Therefore the normalized LMS filter is used to overcome this problem. The different between the LMS and NLMS is the product vector $u(n)e^*(n)$ is normalized with respect to the squared Euclidean norm of the tap-input vector, $u(n)$ as shown in equation (7).

$$w(n+1) = w(n) + \frac{\mu}{||u(n)||^2} u(n)e^*(n) \quad (7)$$

2.7 Recursive Least-Squares (RLS) Algorithm

The recursive-least-squares (RLS) algorithm is developed based on the least squares method [3]. The least-squares method is a mathematical procedure for finding the best fitting curve to a given set of data points. This is done by minimizing the sum of the squares of the offsets of the points from the curve. The summary of RLS algorithm is shown in the equation (8), (9), (10), (11) and table 2.2.

$$k(n) = \frac{P(n-1)u(n)}{\lambda + u^H(n)P(n-1)u(n)} \quad (8)$$

$$e(n) = d(n) - w^T(n)u(n) \quad (9)$$

$$w(n) = w(n) + k(n)e^*(n) \quad (10)$$

$$P(n) = \lambda^{-1}P(n-1) - \lambda^{-1}k(n)u^T(n)P(n-1) \quad (11)$$

The description of the variables is shown in table below

Variable	Description
$k(n)$	The gain vector at time n
$P(n)$	The inverse correlation matrix at time n
λ	Exponential weighting factor
$e(n)$	The estimation error at time n
$d(n)$	Desired response at time n
$u(n)$	M-by-1 tap-input vector at time n, $[u(n), u(n-1), \dots, u(n-M+1)]^T$
$w(n)$	Tap-weight vector at time n
$w(n+1)$	Estimate of tap-weight vector at time n+1

Table 2.2: Description of variables in RLS algorithm equations.

2.8 Feasibility of Fixed-Point Transversal Adaptive Filter in FPGA.

According to Andrew [5], the adaptive algorithm, namely the LMS algorithm, can be implemented based on fixed-point arithmetic in FPGA. Transversal filters have fixed weights and the output of the filters is the convolution of the taps and the filter coefficients. Transversal adaptive filters need an appropriate algorithm to update the filter coefficients and are widely used in the communication industry, as well as in applications such as echo noise cancellation, adaptive beamforming, and channel equalization.

The adaptive algorithm can be implemented in FPGA by using sufficient bit length to represent tap-weights in adaptive filter. However without performing the arithmetic in floating-points, *stalling* may arise in fixed-point adaptation process. This cause the tap-weight stop updated in the arithmetic calculation. But it can be avoided by choosing suitable bit length according to the filter coefficients and study the nature of experiment carefully [3].

2.9 Comparison of DSP processor with FPGA

Nowadays FPGA is becoming a more popular choice for designer to implement flexible and more cost-effective solution with shorter time. There are several companies produce the high performance FPGAs, which have more advantages over DSP processor.

According to Brian Jentz [6], FPGAs have evolved to better support DSP applications, offering the flexibility to implement custom interfaces and peripherals and the capability to scale algorithm complexity and channels as feature requirements grow and change. It provides more than 180,000 Logic Elements (LEs) and 384 18 x 18 multipliers. Enabling Altera FPGA devices to provide 10x the DSP performance per dollar compared to the industry's most widely used DSP processor solutions. To ease comparison, the performance of DSP processor and several FPGA devices [7], [8], [9] are compared in the table 2.3. Obviously Stratix II FPGA from Altera provide the greatest performance with clock speed 450 MHz and 346 Gillion of Multiply Accumulates per second (GMAs).

Type of Device	Device Name	Clock Speed (MHz)	GMACs
DSP	TI DSP	1000	4
FPGA	ECP-DSP20	250	7
	Virtex-4SX55	500	256
	Stratix II	450	346

Table 2.3: Performance comparison between DSP processor and FPGA.

For the current trends, manufacturers include complete microprocessors within the FPGA fabric. This mix of hardware and embedded software on a single chip is ideal for fast filter structures with arithmetic intensive adaptive algorithms.

2.10 Other noise cancellation techniques

Nonlinear blind source separation can be used as a technique for noise cancellation also. Blind source separation aims to recover unobservable independent sources (or signal) from multiple observed data masked by linear or nonlinear mixing. It can be done by using a Radial Basis Function (RBF) network with neural-network approach. Two algorithms are used to develop this RBF network, *stochastic gradient descent method* and *unsupervised clustering method* [10].

In addition, adaptive noise cancelling can be implemented based on independent component analysis (ICA). The ICA-based algorithm can utilize higher order statistics than using least-mean-squares (LMS) algorithm. It is derived to improve convergence rates [11].

2.11 HDL

Hardware description language (HDL) is similar to programming language, which is used to describe digital hardware. The logic diagram, digital information and operation of digital system are represented in textual forms [12]. Therefore it can be read by the human and computer. The HDL processing is used for the simulation and synthesis purpose. A test bench is used to perform the functionality simulation and timing diagram. Hence the designer can correct the error and predict how the designed hardware behave before it is fabricated. While synthesis is the process of deriving a list of components and their interconnections (netlist) from the model of a digital system described in HDL. The netlist can be used to fabricate an integrated circuit or to layout a printed circuit board.

2.12 Verilog

Verilog is a general-purpose hardware description language (HDL) that is easy to learn and easy to use. It is aimed at providing a functionally tested and a verified design description for the target FPGA or ASIC. The functions of this language are fulfilling the need of a design description and the need to verify the functionality and timing constraint of the design [13]. The syntax used in Verilog is similar to the C programming language. It allows different levels of abstraction to be mixed in the same model. Thus the hardware model can be defined in terms of switches gates, Register Transfer Level (RTL), or behavioral code [14].

2.13 FPGA

Field programmable gate array (FPGA) is a VLSI circuit that can be programmed by HDL. A typical FPGA consists of an array of thousands or millions of logic blocks, surrounded by programmable input and output blocks and connected together via programmable interconnections [12]. The logic block consists of look-up tables, multiplexers, gates, and flip-flops. The look-up table is a truth table stored in a SRAM and provides the combinational circuit function for the logic block. The advanced FPGA can consist of 18 bits x 18 bits multiplexers and Configurable Logic Blocks (CLBs).

2.14 Multiplier

The block diagram of common binary multiplier is redraw from [12] as shown in figure 2.4. The multiplicand is stored in the register B and multiplier is stored in the register Q, and the partial product is formed in register A and stored in A and Q. The data in the register A is added with register B by parallel adder. The value of carry bit after the summation is stored in the C flip-flop. The P counter holds the number of bits in the multiplier initially.

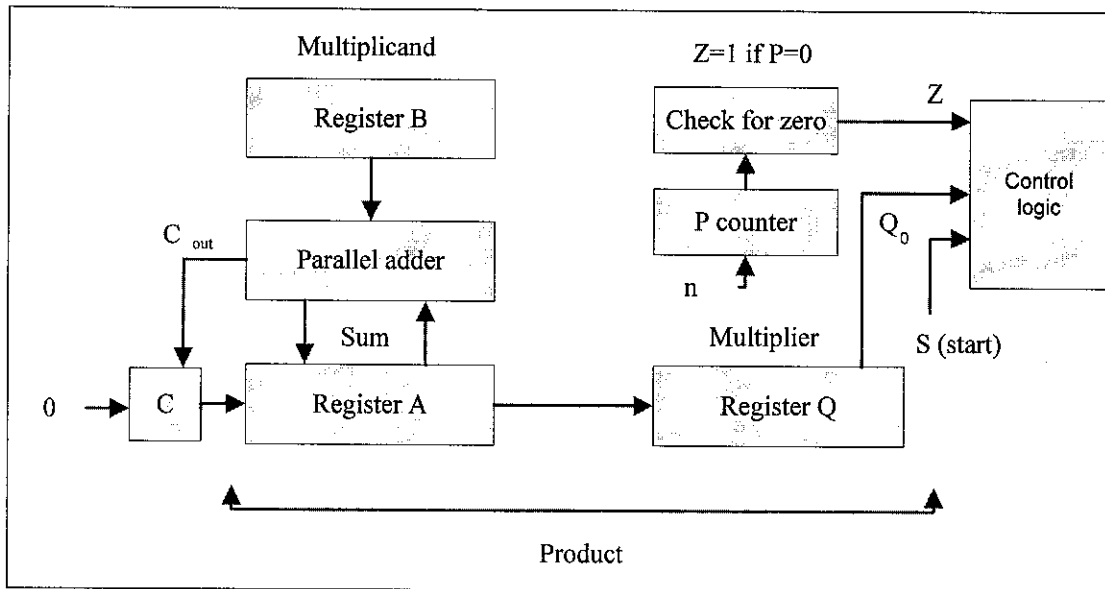


Figure 2.4: Block diagram of binary multiplier

(Source: M. Morris Mano, 2002. Digital Design. Prentice Hall, Inc.)

It is decremented after the formation of each partial product. The product is formed in the combination of register A and Q when the counter reaches zero, and the process stops. When $S=1$, the control logic start performs the multiplication. The sum of A and B is stored back into A, and the output carry from the addition is transferred to C. This bit is shifted to the MSB of register A after the summation, while the LSB of register A is shifted to the first bit of register Q, and 0 is shifted into C. After the shift right operation, the LSB of register Q (Q_0) is send to the control logic. This bit is used to determine whether to add or not. However this type of multiplication architecture is not suitable for multiplication of signed number in the filter design.

2.15 Multiplication of Signed Binary Number

Booth's algorithm and Bit-Pair Encoding can be used for the multiplication of signed value [15]. If the signed number for multiplication has a negative value, the multiplication result is not the same as the normal calculation with calculator. For example, $-7 \times 7 = -49$ can be implemented step by step with serial multiplier [16] as calculation below:

Sign extension

		1001 (-7)
		<u>0111 (7)</u>
→		11001
		<u>1001</u>
↘		1101011
		<u>11011</u>
		11001111 (-49)

Below are another two examples with multiplication of two 16 bits numbers, one is involving negative value (f001 x 0005) and another one with positive value only (100a x 0005).

f001 x 0005:

```

1111 0000 0000 0001 (f001)
      101 (0005)
      -----
111111 0000 0000 0001
+1111 0000 0000 0001
-----
1111 1111 1111 1111 1011 0000 0000 0101 (ffffb005)

```

100a x 0005:

```

          0001 0000 0000 1010 (100a)
          _____
                101 (0005)
          000001 0000 0000 1010
          +0001 0000 0000 1010
          _____
0000 0000 0000 0000 0101 0000 0011 0010 (00005032)

```

2.16 Multiplication of Fractions Number

The fixed-point integer value is converted to the fixed-point fractional value by normalization, which divides an N-bit 2s complement word with 2^{N-1} . The normalize value for a 2s complement word B with N bits is $N(B) = -b_{N-1} 2^0 + b_{N-2} 2^{-1} + \dots + b_1 2^{-(N-2)} + b_0 2^{-(N-1)}$. [15] For the filter design, the input and output data are represented in fixed-point fractional number. The data is represented with 16 bits, with the first bit as the integer number and the following bits as fractional number. Therefore it can represent number ranges from -1 to 1.

To ease understanding, figure 2.5 shows how to represent decimal value -0.02130126953125 in fractional number. The decimal value can be determined by adding up all the value according to equation shown in [15]. $N(B) = -2^0 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-13} + 2^{-14} = -0.02130126953125$. This value is represented in hexadecimal format as fd46, which is one of the coefficients of FIR filter. The method to perform multiplication of fraction number is same as multiplication of signed number as discussed in section above. But it is important to know that the number for the calculation is represented with how many integer bits and fraction bits.

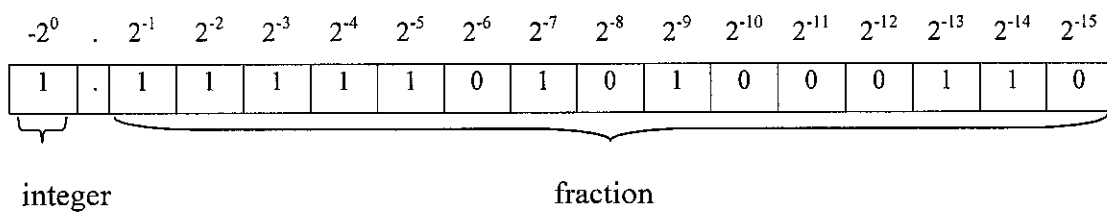


Figure 2.5: Representation of decimal value -0.02130126953125 in fractional number

CHAPTER 3

METHODOLOGY/PROJECT WORK

The methodology of the project can be divided into four main parts, which include research, Matlab simulation, ModelSim simulation and hardware implementation with FPGA as shown in figure 3.1.

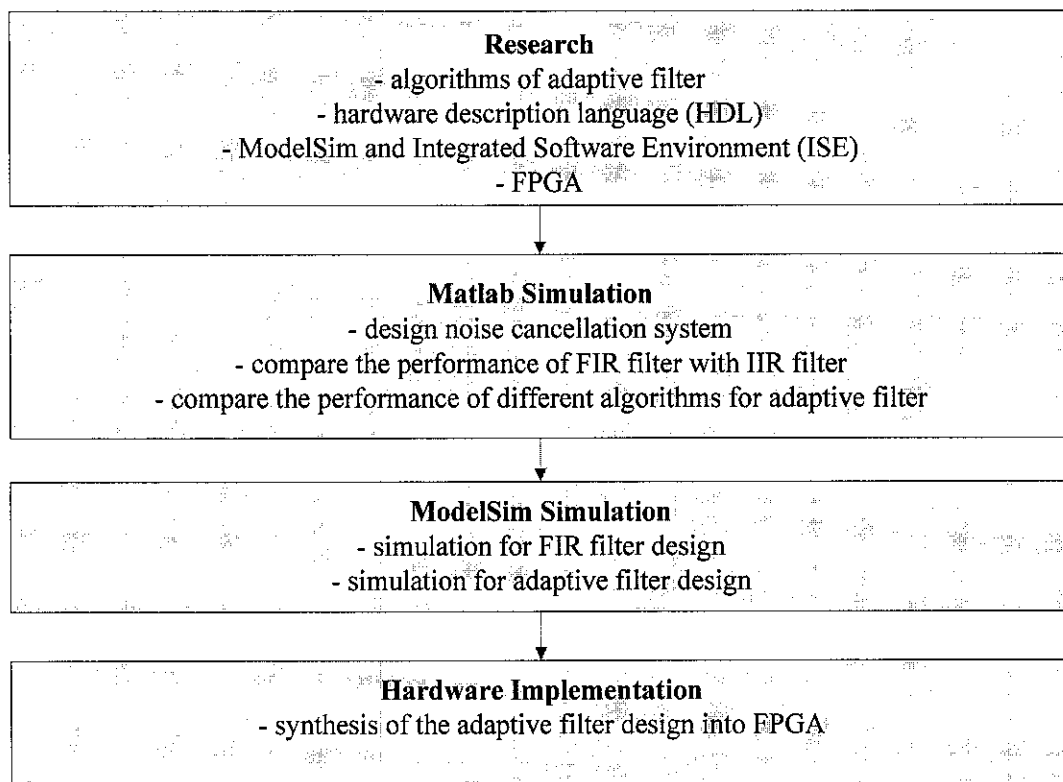


Figure 3.1: Methodology of adaptive filter design

3.1 Research:

The project starts with research on several noise cancellation techniques, which cover the electrical noise, acoustic noise and other noises occur in the communication system. The techniques used including shielding, grounding and twisting of wire for electrical noise. While for the acoustic noise and other noises in communication, the techniques used are active noise control and adaptive

filtering. Furthermore the research also includes the algorithms used in adaptive filter, Hardware Description Language (HDL), simulation software of HDL, FPGA program software and the advantages of using FPGA over DSP processor. Most of the research was on adaptive filter that use the Least-Mean-Square algorithm for noise cancellation.

3.2 Matlab Simulation:

This section will describes how Matlab Simulink and Filter Design Analysis (FDA) tools are used to analysis the performance of the filter design and to implement adaptive filter. These include performance comparison between the direct form FIR filter and direct form IIR filter in designing adaptive filter with different algorithms (LMS, NLMS and RLS). Furthermore adaptive noise cancellation system is designed and simulated in the Simulink.

3.2.1 Filter and Algorithm Determination

In the adaptive noise cancellation system, the selection of the filter and algorithm will affect the overall performance of the system. Therefore the combination of the FIR filter or IIR filter with LMS, NLMS or RLS is tested in the Simulink. However before starting design, it is necessary to select the proper input to test the system, using the Filter Design Analysis (FDA) tool to design filter, and implement the algorithms used in adaptive filter.

For better comparison, the output waveform and sound from the adaptive noise cancellation system is observed. Figure 3.2 shows the Simulink example for waveform observation and figure 3.3 for sound observation. The recorded music is used as original signal to the system, with a sampling frequency of 8000 Hz. The chirp signal is chosen as a noise to test this noise cancellation system because chirp signal produces noise that cover large range of frequency, as time increase, its frequency also increase, therefore it is suitable for the above purpose. The Matlab code in APPENDIX B shows how to generate the input signal and chirp noise. In Matlab code, `WAVRECORD(N, FS, CH)` records N audio samples at FS

Hertz from CH number of input channels from the windows WAVE audio device.

While `Y=CHIRP(T,FO,T1,F1,'quadratic',PHI,'convex')` generates samples of a quadratic swept-frequency signal whose spectrogram is a parabola with its convexity in the positive frequency axis.

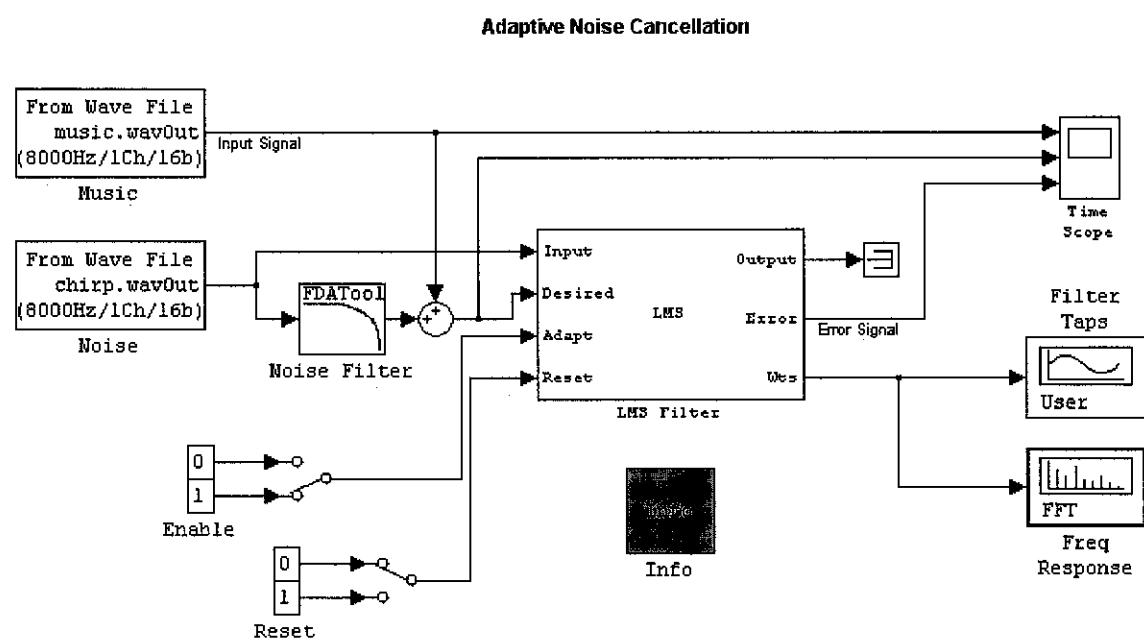


Figure 3.2: Simulink for waveform observation with FIR filter and LMS algorithm

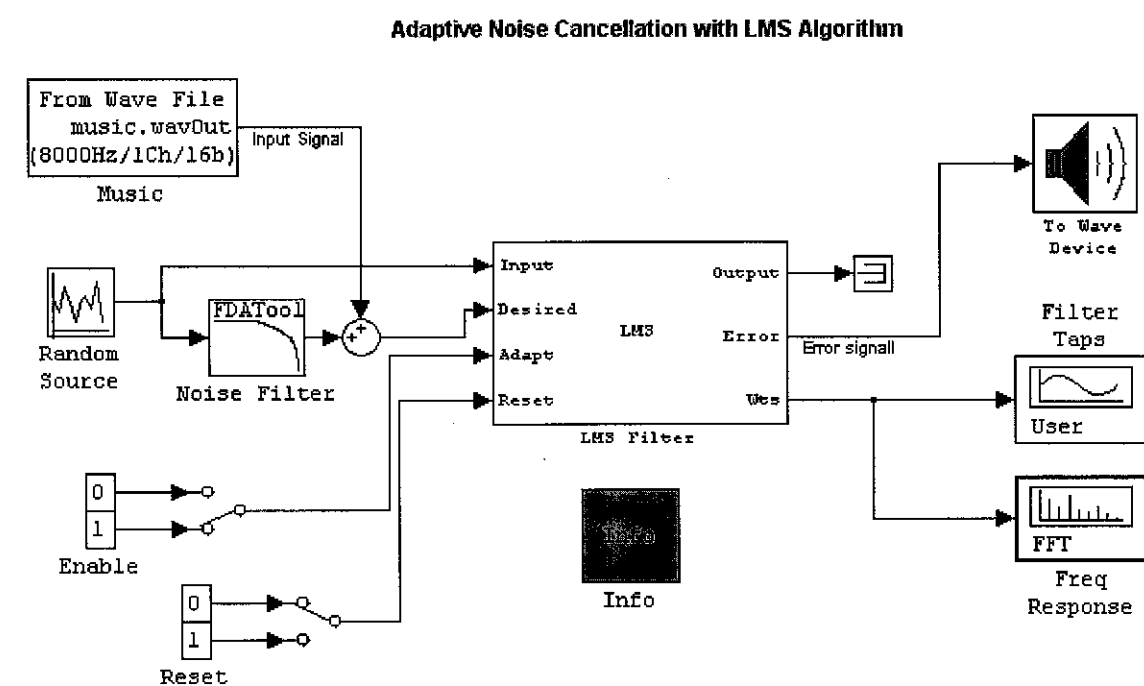


Figure 3.3: Simulink for sound observation with FIR filter and LMS algorithm

The direct form FIR filter and direct form IIR filter can be easily designed with FDA tool in Matlab as figure 3.4 shown. Both filters are of 25 orders with cut off frequency approximate at 2000 Hz. Since IIR filter can use fewer orders to get the similar spectra characteristics as FIR filter, the comparison also include the 10 orders IIR filter.

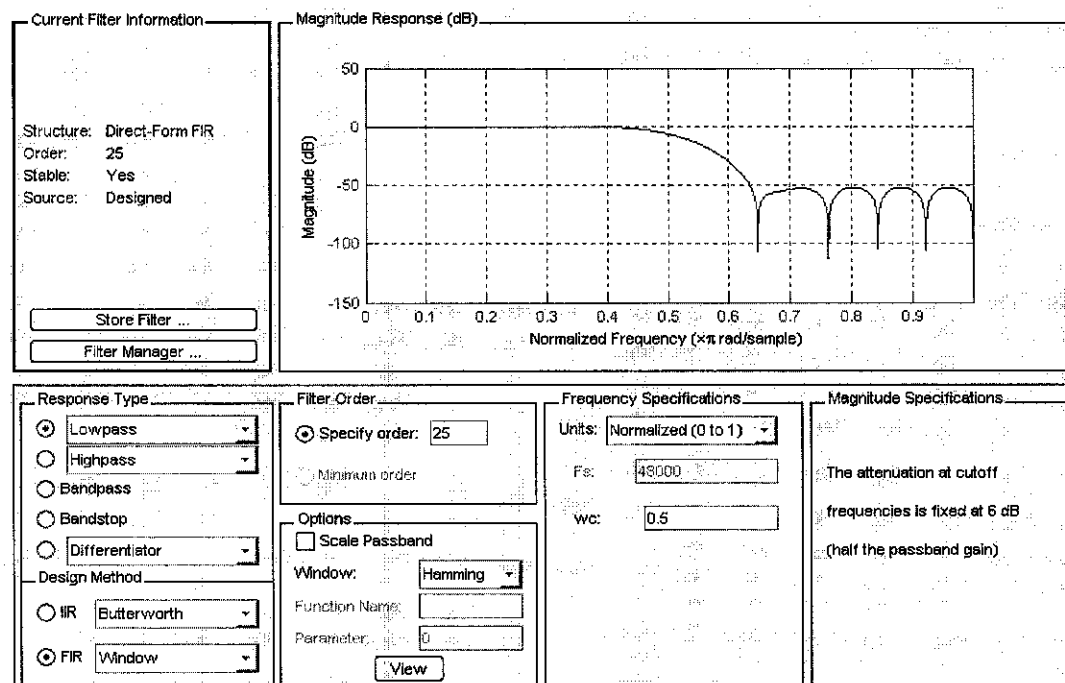


Figure 3.4: Filter design with FDA tool

Besides that, the adaptive noise cancellation system can be implemented with the Matlab code. It is assumed that two microphones are used, a primary microphone picks up the noisy input signal, while a secondary microphone receives noise that is uncorrelated to the information of an original signal (0.055Hz sine wave), but is correlated to the noise picked up by the primary microphone. The Matlab code in APPENDIX C shows how the Adaptive LMS filter extract useful information from a noisy signal.

3.3 ModelSim Simulation

ModelSim XE-III is used for the Hardware Description Language (HDL) simulation of adaptive filter design in this project. ModelSim is a complete HDL simulation environment that assists to verify the HDL source code and functional

and timing models of the designs. Each of the ModelSim tools includes a complete HDL simulation and debugging environment providing full VHDL and Verilog language coverage, a source code viewer and editor, waveform viewer, design structure browser, list window, and a host of other features designed to enhance productivity.

After the result of Matlab simulation is satisfied with the objective of the project, the Verilog code for the design will be written and simulated with the ModelSim before implemented with Xilinx Integrated Software Environment (ISE). The general steps for ModelSim simulation are summarized as below.

- 1.) Collating design file and mapping libraries
- 2.) Compiling the design with *vlog*, which compiles Verilog source code into a specified working library
- 3.) Loading the design for simulation, VSIM simulator is invoked.
- 4.) Simulating the design and viewing the waveform of the design.

Since the FIR filter is the most important part in implementing adaptive noise cancellation, the HDL code for the filter should be completed before implementing the LMS algorithm on it. Figure 3.5 is the FIR filter structure for a 10 order direct-form filter, which consists of 11 coefficients, 11 adder and 10 delays. Blackman window is chosen for the filter design. This is because it provides more attenuation at the stopband frequency if compared with Hamming window and Hann window. The Verilog code for a 10 order direct-form FIR filter is shown in APPENDIX D.

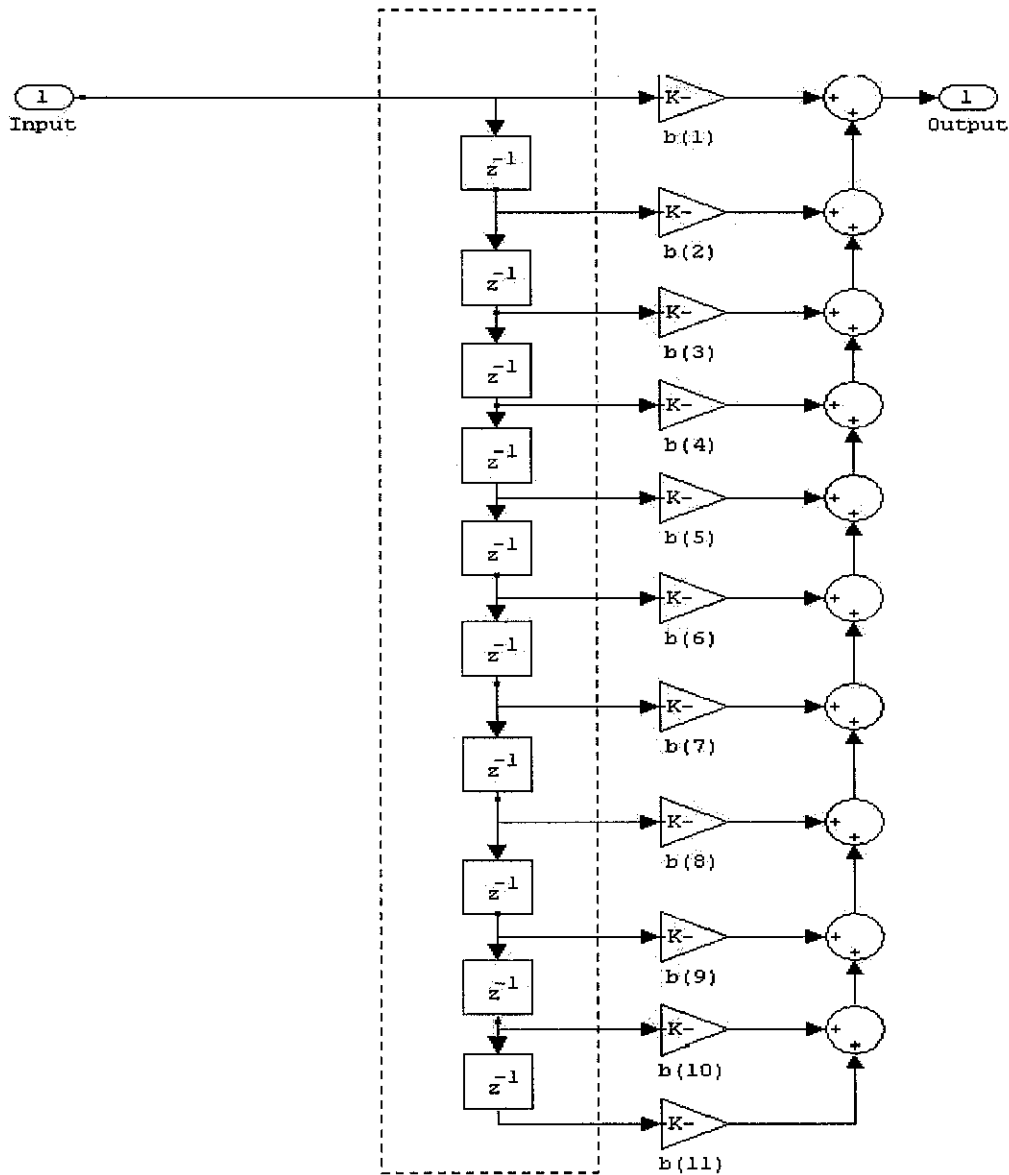


Figure 3.5: 10 orders FIR filter structure generated from Matlab

The flow of the filter design in Verilog coding is shown as figure 3.6. The first step is defines the filter coefficients get from FDA tools and declares the required scalars or vectors for input, output and arithmetic computation. If the reset is in logic HIGH, the input and output data of filter is set to zero. Otherwise the input data in the testbench is loaded to the filter. For every positive edge of the clock signal, the input data is shifted one by one until the end of the data. This constructs the delay part of the filter. After that, the output data from the delay is multiplied with the coefficients of the filter. The multiplication results are summed to produce an output data. In every summation, program examines the overflow bit of data. If the data is overflow after summation, the data is saturated

to produce the data that is in the range of defined fixed-point value. Then it is round to the 16 bits data before assign to the output port.

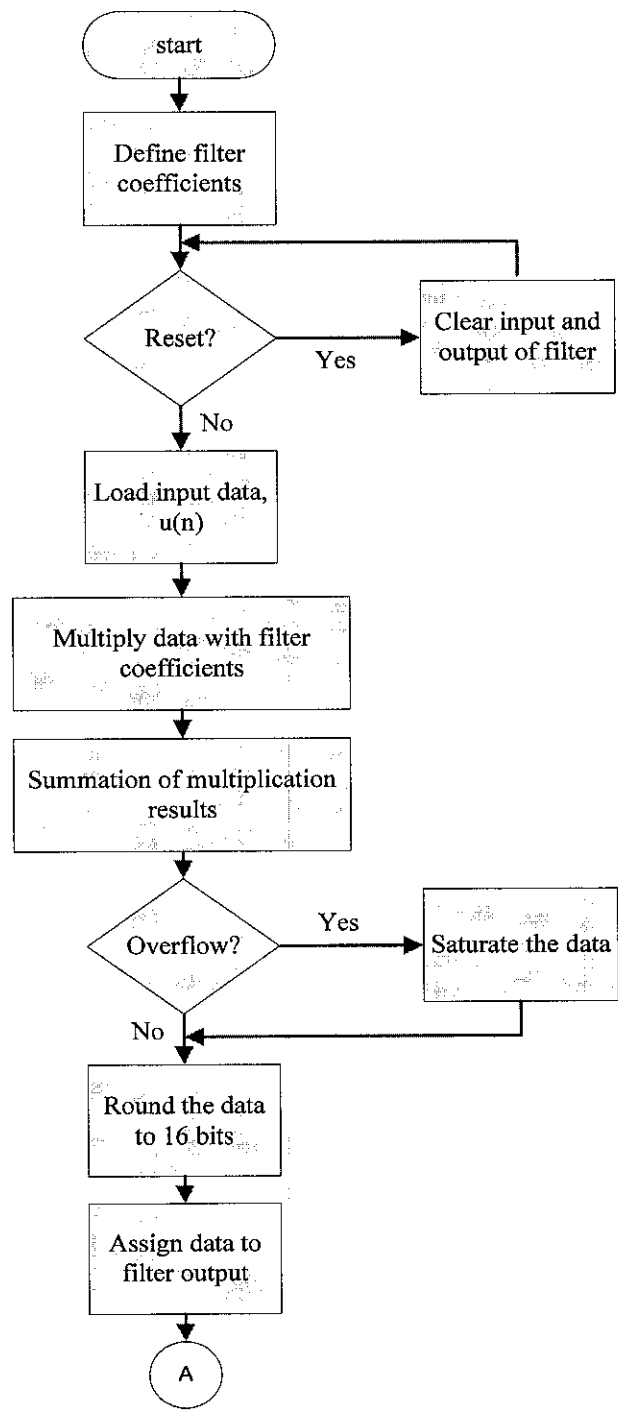


Figure 3.6: Flow chart of filter design in Verilog

The flow chart in figure 3.6 is extended for the implementation of the adaptive algorithm. The adaptive filter design is illustrated in figure 3.7. The desired data in the testbench is loaded and the error is calculated by using equation (4) that had mentioned before. Then equation (5) is used to calculate and update the weight of adaptive filter. The adaptive filter is continuously calculating the error and updates the weight until end of data. Please refer to the APPENDIX E for the coding of adaptive filter.

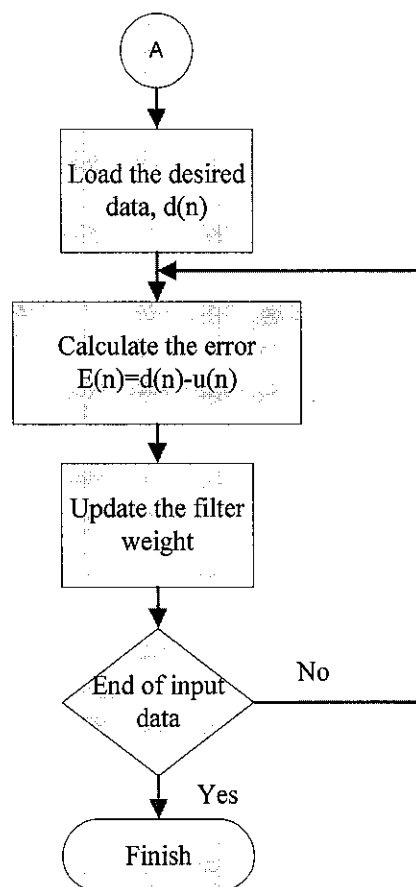


Figure 3.7: Extended flow chart of adaptive filter design

3.4 Hardware Implementation

3.4.1 Tools used

ISE software

The Integrated Software Environment (ISE) is the Xilinx design software for logic design environment. It is an easy-to-use software that provides built-in tools and wizards for making I/O assignment, power analysis, timing-driven design closure, and high speed HDL simulation. It supports all Xilinx leading FPGA and Complex Programmable Logic Device (CPLD), including all Virtex-4 multi-platform FPGAs.

Virtex-II FPGA

The Virtex-II family is a platform FPGA developed for high performance, low to high-density designs utilizing IP cores and customized modules [17]. It delivers complete solutions for telecommunication, wireless, networking, video, and DSP applications. Virtex-II devices are user-programmable gate arrays with various configurable elements, which comprised of input/output blocks (IOBs) and internal configurable logic blocks (CLBs). In addition, its dedicated 18-bit x 18-bit multiplier blocks arithmetic functions, which assist the implementation of adaptive filter with complex calculation.

Virtex-II Development Kit

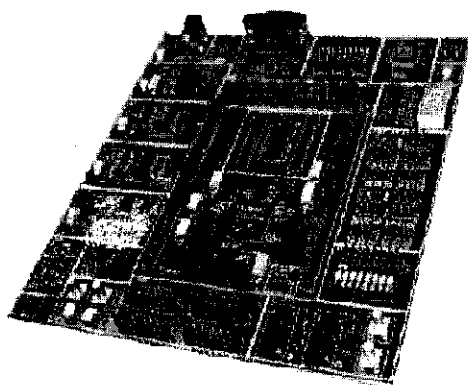


Figure 3.8: Virtex-II Development Kit

The filter design is implemented with the Virtex-II Development Kit shown in figure 3.8. This reference board provides a development platform for prototyping and verifying Virtex-II based designs [18]. With utilization of 1M gate Virtex-II XC2V1000-4FG256C device along with its supporting I/O devices, it can be used to prototype high-performance memory and I/O interfaces such as complete high-performance low voltage differential signaling (LVDS) and high speed DDR memory interface. Therefore it is chosen for the implementation of the adaptive filter.

Xilinx XC18V512 or XC18V04 ISP PROM is utilized in this board. It allows the design downloaded and verified in order to meet the final system-level design requirements. In addition, the Joint Test Action Group (JTAG) connector of the board can be used for direct configuration of the Virtex-II FPGA.

3.4.2 Design Flow

This section describes the procedures in hardware implementation of FIR filter that had simulated successfully in the ModelSim. The general FPGA design stages in [19] are used in FPGA implementation as shown in figure 3.9.

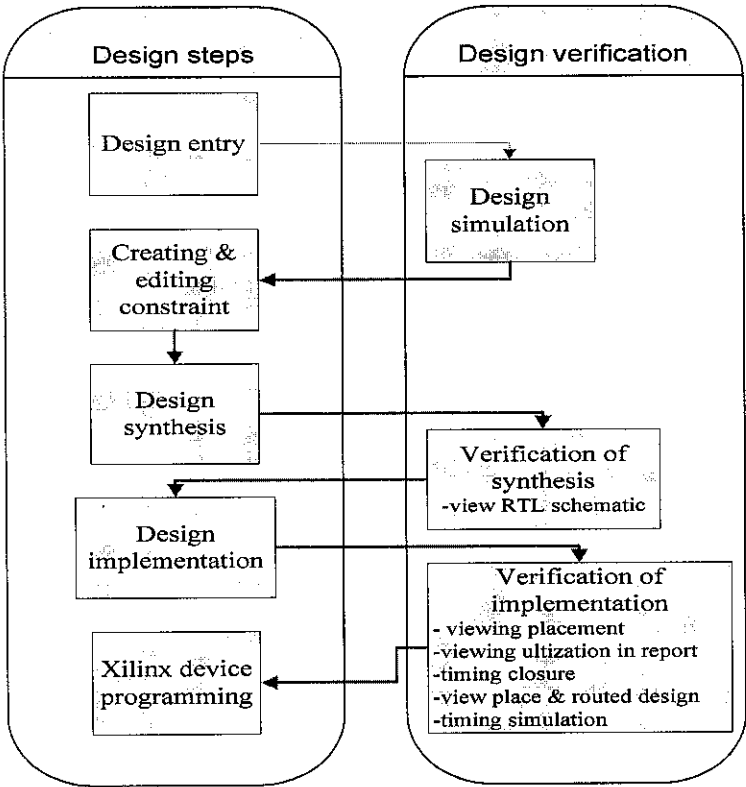


Figure 3.9: General FPGA Design Stages.

In design specification stage, the design is transfer to HDL with Verilog. Then the functionality of written code is simulated. A test bench is needed to provide the necessary input or stimulus to the filter. The output of the filter will be shown as digital waveform. After the design has been successfully analyzed, the next step is to translate the design into gates and optimize it for the target architecture, which called synthesis phase. The design will be implemented by using Virtex-II FPGA from Xilinc. The performance of the FPGA is shown in the APPENDIX F.

3.4.3 Design Entry

This is the first step in the hardware implementation. The source file is created with Verilog language based on the design objective. Type of devices, package, top-level module type, synthesis tool and simulator is specified as below:

Device Family	:	Virtex II
Device	:	xc2v1000
Package	:	ft256
Speed Grade	:	-4
Top-Level Module Type	:	HDL
Synthesis Tool	:	XST
Simulator	:	ISE Simulator (or ModelSim)

3.4.4 Design Simulation

A test bench is needed to provide the necessary input or stimulus to stimulate the filter module. It is used with a simulator to verify that the filter design meets both behavioral and timing design requirements. By using ISE foundation, the design can be simulated with ISE simulator or ModelSim simulator. However ModelSim simulator is prefer because it allows the user to view the waveform in analog form with analog interpolated function in format menu. The test bench can be created by using waveform editor in ISE foundation or wrote by ourselves.

3.4.5 Creating and Editing Timing and Area Constraints

ISE software allow user to specify the constraints to improve the design performance. Timing constraints is used to assure that physical and timing requirements are met. Timing constraints include period constraints for each clock (PERIOD), setup times for each input (OFFSET_IN), and clock-to-out constraints for each output (OFFSET_OUT). The timing constraints can be entered using the Create Timing Constraints process in Project Navigator.

In addition to timing constraints, physical constraints are added to filter design, to associate certain pins on the device with specific inputs and outputs. There are totally 16 input pins and 16 output pins for this filter design. Both constraints processes have written into the User Constraints File (UCF) in the project.

3.4.6 Design Synthesis and Implementation

After the design's behavior is verified with simulation, and added constraints, the design is synthesized and implemented. According to the ISE Quick Tutorial Version 7.1i [19], with Xilinx Synthesis Technology (XST) in ISE software, the Verilog code of filter design is synthesized to create Xilinx-specific netlist files known as NGC files, which consists of an Electronic Data Interchange File (EDIF) with an associated Netlist Constraint File (NCF). The synthesized design can be viewed as a schematic in the Register Transfer Level (RTL) Viewer. The schematic view shows gates and elements independent of the targeted Xilinx device.

The design implementation comprised of the following steps [19]:

1. *Translate*, which merges the incoming netlists and constraints into a Xilinx design file
2. *Map*, which fits the design into the available resources on the target device
3. *Place and Route*, which places and routes the design to the timing constraints

3.4.7 Verification of the Implemented Design

After implementation is completed, the filter design is verified before downloading to an FPGA. The steps include viewing of pin placement, viewing of resource utilization in reports, timing closure, viewing of placed and routed design and timing simulation.

Viewing Placement and Resource Utilization in Reports

Floorplanner is used to verify pinouts and placement of the filter design [19]. The connection from the gates to output pins can be view by clicking on the desired pin. The filter design information is check through summary reports, which was created by ISE after each process is run.

Timing Closure

Timing closure is the process of working on design to ensure that it meets necessary timing requirements. Timing analysis is run on filter design to verify that timing constraints were met [19]. There should be no error in the timing summary after analysis. If there were error, the previous timing setting should be adjusted until no error is shown in timing summary.

Viewing the Placed and Routed Design

FPGA Editor is used to view the filter design on the FPGA device, as well as edit the placement and routing with the FPGA Editor [19]. Figure 3.10 is the placement and routing of an output pin.

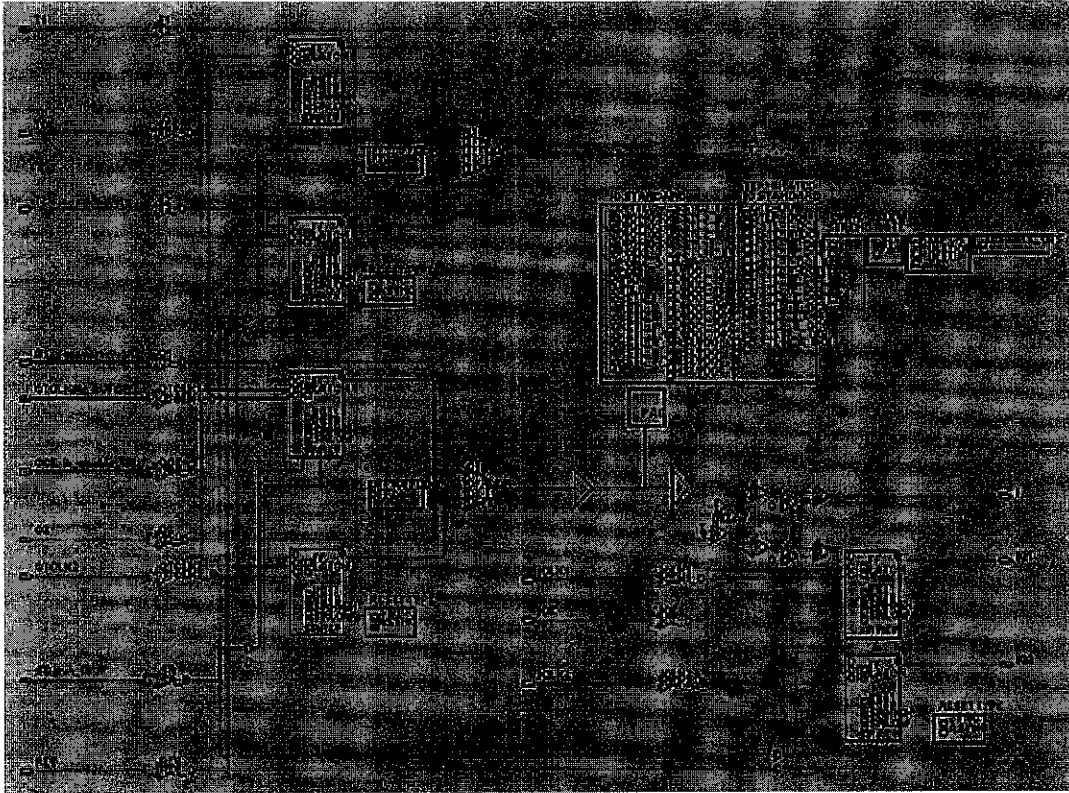


Figure 3.10: FPGA Editor - Detailed view of filter design for an output pin.

Timing Simulation (ISE Simulator)

Timing simulation is run to verify that the filter design meets the timing requirement. This process generates a timing-annotated netlist from the implemented design and simulates it [19]. The resulting simulation is displayed in the Waveform Viewer. However these results look different from those saw in the behavioral simulation. These results show timing delays.

3.4.8 Creating Configuration Data

The final phase in the software flow is to generate a bitstream and configure the device. The bitstream is a binary encoded file that is the equivalent of the design in a form that can be downloaded into the FPGA device. iMPACT is used to configure the FPGA device [19].

CHAPTER 4

RESULTS AND DISCUSSION

The results and discussion is divided to 3 sections: Matlab Simulation, ModelSim Simulation and Hardware Implementation.

4.1 Matlab Simulation

This section includes the result and discussion for comparison on the performance of the FIR filter and IIR filter with different algorithms, the output of adaptive noise cancellation with Matlab code and some discussion with the results.

Through the Simulink simulation (figure 3.2 and figure 3.3), the performance of filter output can be compared with the graph and sound from the simulation. The graph of filter output for the simulation in figure 3.2 is included in APPENDIX G, and the results are summarized in table 4.1.

The results in table 4.1 show that RLS algorithm has better results than other two algorithms when comparing the output of the filter in graphs, since it removes most of the noises in shorter time. But for the output sound of the filter, NLMS performs better. In contrast, RLS is a bit lagging in producing output sound, which means the output sound cannot be heard clearly and smoothly. This is because more complex calculations are required with RLS algorithm.

While for the filter analysis, 10 orders IIR filter produces similar result as 25 orders FIR filter. However 25 orders IIR filter produces less desirable results, more noise is associated with the output signal. Hence the orders of the filter must be well-adjusted when using the adaptive algorithm.

Algorithm	Filter	Order	Filter output	Audible outcome
NLMS	FIR	25	<ul style="list-style-type: none"> output almost same with the input signal after delay time 0.15 sec 	<ul style="list-style-type: none"> output sound heard similar to the input sound after 0.15 sec
	IIR	25	<ul style="list-style-type: none"> even after delay time (0.2 sec), still got a little noise 	<ul style="list-style-type: none"> output signal still got audible noise after delay time
		10	<ul style="list-style-type: none"> similar to FIR filter with NLMS 	<ul style="list-style-type: none"> similar to FIR filter with NLMS
LMS	FIR	25	<ul style="list-style-type: none"> output signal not same as input signal, has little noise. 	<ul style="list-style-type: none"> audible noise in the output signal
	IIR	25	<ul style="list-style-type: none"> more noise than using FIR filter 	<ul style="list-style-type: none"> can clearly hear the noise, louder than using FIR filter
		10	<ul style="list-style-type: none"> similar to FIR filter with LMS 	<ul style="list-style-type: none"> similar to FIR filter with LMS
RLS	FIR	25	<ul style="list-style-type: none"> output signal almost same as the input signal, shorter delay time (0.1 sec) compared with the NLMS 	<ul style="list-style-type: none"> output signal a bit lagging due to more complex calculation compared with the LMS and NLMS. audible noise in the output signal
	IIR	25	<ul style="list-style-type: none"> more noise than using FIR filter with RLS 	<ul style="list-style-type: none"> output signal a bit lagging due to more complex calculation compared with the LMS and NLMS. Loud noise in output signal
		10	<ul style="list-style-type: none"> similar to FIR filter with RLS 	<ul style="list-style-type: none"> audible noise in the output signal

Table 4.1: Summary of the observation from the adaptive noise cancellation system

Therefore NLMS algorithm and 10 orders IIR filter is desirable used for adaptive noise cancellation. This is because with NLMS algorithm, more noise is reduced and there is no lagging at the output sound. Besides that, with 10 orders IIR filter,

there is less calculation compared with the 25 orders FIR filter for similar output result.

On the other hand, an IIR filter may not be suitable for used in hardware implementation. This is because the IIR filter in Matlab simulation is using floating point precision for the coefficients of filter and input output port. When implement in FPGA, the number of output is limited to 16 bits only. This causes the round off errors at the output signal. Since IIR filter involves both feedforward and feedback. That means the error at the filter output is fed back to the input, and the error is accumulated in the system [15].

Adaptive Noise Cancellation System with Matlab Code

As mentioned in the methodology, the adaptive noise cancellation system can be implemented with Matlab code also. With the Matlab code in APPENDIX C, several graphs are generated to show the filter response, original signal, noisy signal and output signal from the system. Figure 4.1 shows the frequency response of the filter. When running the Matlab code, the filter will keep on changing its coefficient value to minimise the error at the output.

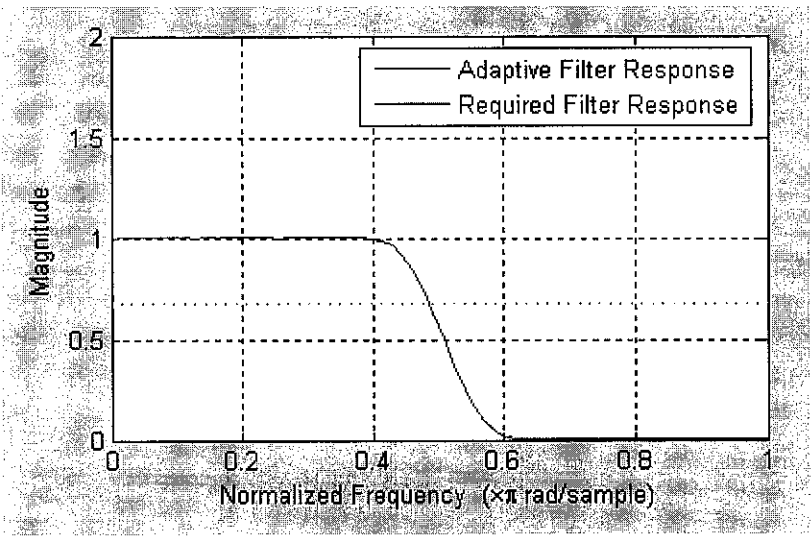


Figure 4.1: Adaptive filter response

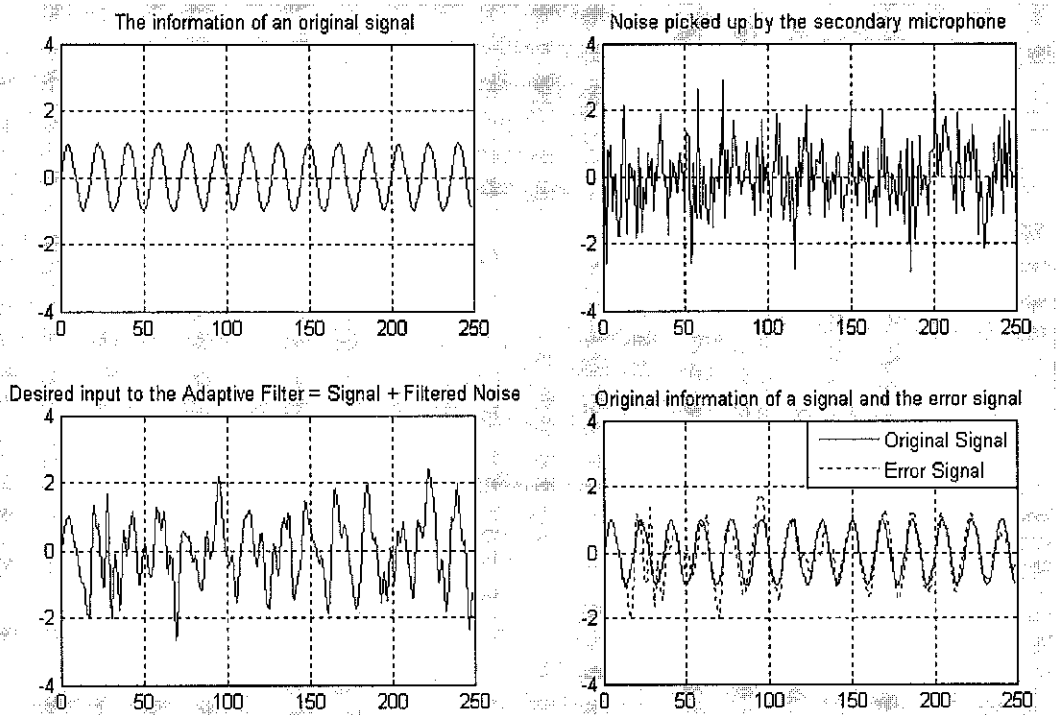


Figure 4.2: The original signal, noise, desired signal and the error signal

From figure 4.2, you can see the original signal is a sinusoidal signal with frequency 0.055Hz. While the noise pick up by the secondary microphone is a white noise, which interferes with the sinusoidal signal. After passing through the adaptive filter, the noise is subtracted from the signal.

4.2 ModelSim Simulation

4.2.1 FIR Filter

ModelSim can perform the functional simulation and timing simulation. The result of functional simulation is discussed in this section, while timing simulation is discussed in hardware implementation section. Through the ModelSim simulation on the Verilog code of filter in APPENDIX D, the desired result is obtained. Figure 4.3 shows the result of functional simulation for FIR filter.

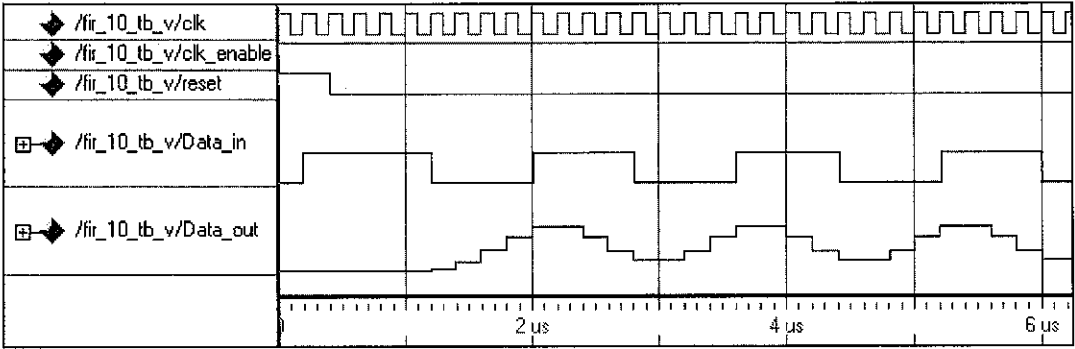


Figure 4.3: Result of functional simulation with ModelSim

From the figure 4.3, the input signal to the filter is a square wave. After the filtering, the high frequency signal is removed and a sinusoidal signal is obtained. However the output signal is not smooth due to quantization error and round off error. This is because when the filter is designed in fixed-point precision, 16 bits are not sufficient to generate a smooth sinusoidal wave.

To distinguish between the fixed-point precision with double precision, a similar filter with double precision is used to test its effect on the output. Figure 4.4 shows the input and output of the filter, which created with Matlab code. Obviously the output from the double precision filter is smoother because 64 bits produce better precision than fixed-point precision filter.

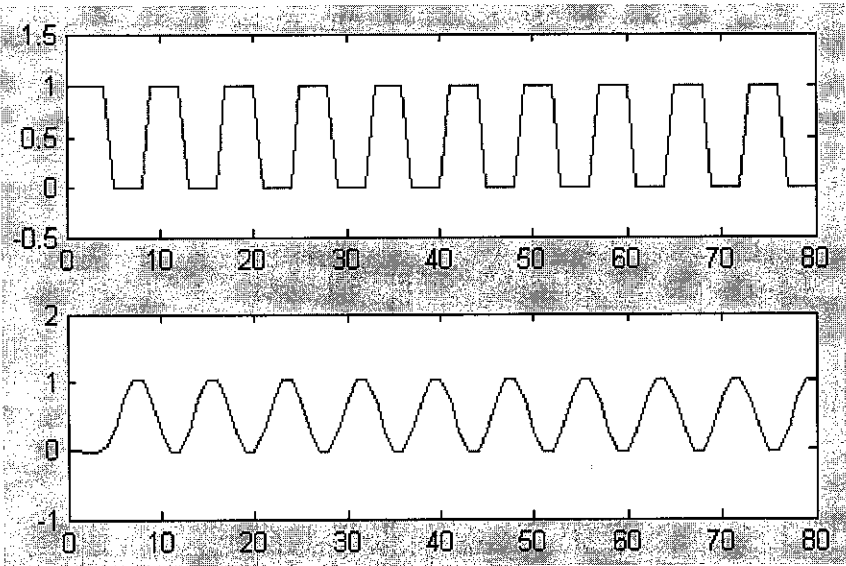


Figure 4.4: Input and output of the filter in Matlab

Fixed-point Precision Effects

The floating point precision can represents any number between +/- 9.223x 10¹⁸ with a resolution of 1.08 x 10⁻¹⁹. However the fixed point precision can only represents the number with smaller range, which depends on the number of bits used to represent integer number and fraction number. The implementation of floating point arithmetic in FPGA is possible [20]. However only a small number of floating point units can be used in an entire design, and must be shared between processes. This does not take full advantage of the parallelization that is possible with FPGAs. Therefore it is not efficient or realistic for implementing adaptive algorithm in FPGA.

As a result, all calculation is mapped to fixed point number. Sixteen bits number is used to represents the coefficient number, input and output data. However it introduces some errors into the design as discussed below. [15]

1. *Coefficient quantization error*

This is due to representation of filter coefficients by a finite number of bits. The coefficients generated from the Matlab are in 64 bits representation. For example the coefficients of Blackman Window FIR filter are shown as figure 4.5. However the smallest resolution for a 16 bits number with 15 bits fractions is 2⁻¹⁵. Therefore the coefficients are quantized to the decimal value and hexadecimal value as shown in figure 4.6.

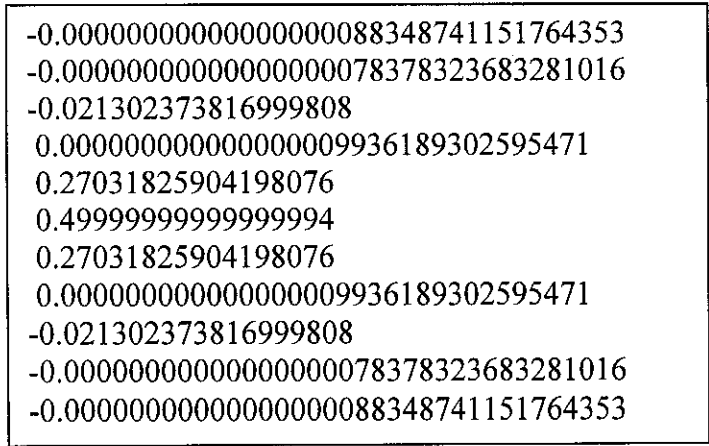


Figure 4.5: The coefficients of filter in floating-point precision

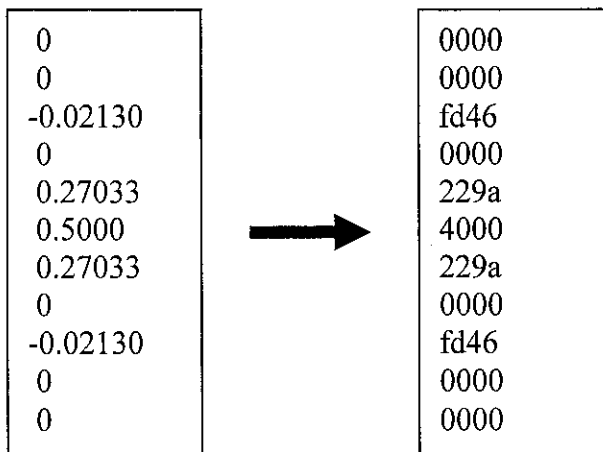


Figure 4.6: The coefficients of filter in fixed-point precision

2. *Overflow error*

This is due to the addition of two large numbers of the same sign which produces a result that exceeds permissible word length.

3. *Round off error*

This is caused when the result of a multiplication is rounded to the nearest discrete value or permissible word length. The result of multiplication in the filter in 32 bits data, however the output data is limited to 16 bits, by discarding the least significant 16 bits, round off error is introduced into the data.

4.2.2 Adaptive filter

For the testing of adaptive filter design in the APPENDIX E, a step input is feed to the filter. If the desired signal is set such that it is the same as the expected output signal, the calculated error is zero as shown in figure 4.7 when implemented with equation (4).

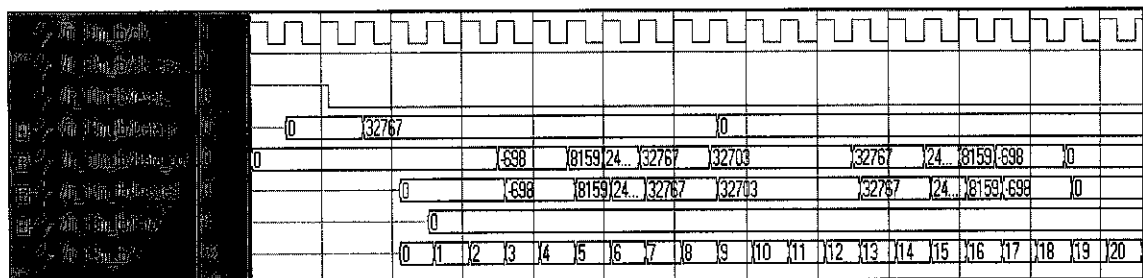


Figure 4.7: Output of filter with zero error.

For the implementation of equation (4) and (5), a pulse signal labelled as ‘Data_in’ is noise. It is added to the step input to produce a ‘Desired’ signal as shown in the figure 4.8. (Please refer back to figure 2.3 for clearer understanding) The output of the filter is ‘Data_out’, which has the quantization error that is same as the output of FIR filter in figure 4.3. Though the implementation of adaptive filtering, the expected output of adaptive filter ‘Error’ is a step signal with noise initially. After a few loops of weights update, the error should be reduced to nearly zero. This means the pulse signal is subtracted from the ‘Desired’ signal. As shown in figure 4.8, the error is large in ‘Error’ signal initially. After nearly 300 ns, the error is greatly reduced after the tap-weights of adaptive filter are updated for some loops. However the error in this design is not reduced to zero. This is caused by the effects of fixed-point precision, which including the quantization error, overflow error and round off error as mentioned before. As a result, the small error is repeated in the ‘Error’ signal, which is highlighted by the dotted box as shown in figure 4.8.

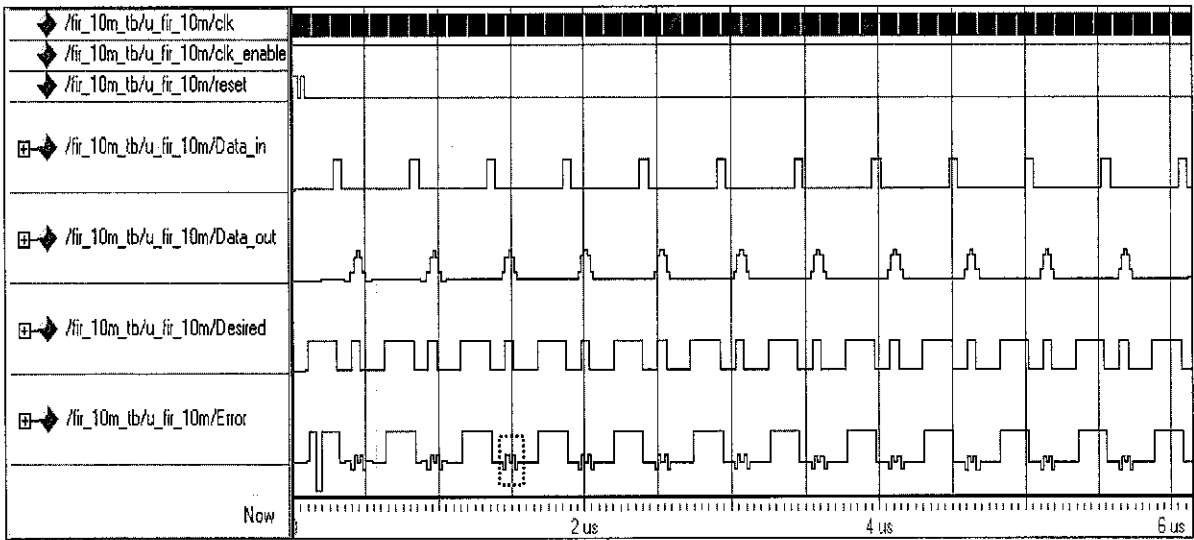


Figure 4.8: Output of adaptive filter with error

A lot of round off errors is introduced to the adaptive filter when implement the LMS algorithm with equation (5). This equation involves two multiplications and a summation in mathematic calculation. These multiplications will generate a bit stream with 48 bits if it is multiplied directly. In addition, Virtex II FPGA only

can provide 18 bits x 18 bits multiplications in hardware. Therefore the multiplications are done in several stages before the summation.

Figure 4.9 shows part of the coding for implementation of equation (5) in the algorithm of adaptive filter. The result of multiplication of equation (5) is stored in the *mumu1*. For updating the coefficients of adaptive filter, 16 MSB bits of *mumu1* is stored in the *mumu2*. Then *mumu2* is used to update the weights of the adaptive filter. This produces the result as shown in the figure 4.8.

```

mult1 = (Data_in*error_m);
temp1 = mult1[31:16];
mumu1 = mu*temp1;
mumu2 = mumu1[31:16];
c[0] = c[0]+ mumu2;
c[1] = c[1]+ mumu2;
c[2] = c[2]+ mumu2;
c[3] = c[3]+ mumu2;
c[4] = c[4]+ mumu2;
c[5] = c[5]+ mumu2;
c[6] = c[6]+ mumu2;
c[7] = c[7]+ mumu2;
c[8] = c[8]+ mumu2;
c[9] = c[9]+ mumu2;
c[10] = c[10]+ mumu2;

```

Figure 4.9: Verilog coding for implementation of equation (5)

4.3 Hardware Implementation

In this section, two important simulations performed are functional simulation and timing simulation. Where functional simulation is done by running Simulate Behaviour Model and timing simulation is by running Simulate Post-Place & Route Verilog in the ModelSim Simulator of ISE. For the FIR filter and adaptive filter design, the result of functional simulation is discussed previously in section 4.2 ModelSim Simulation, while the result of timing simulation is shown in figure 4.10 for FIR filter and in figure 4.11 for adaptive filter.

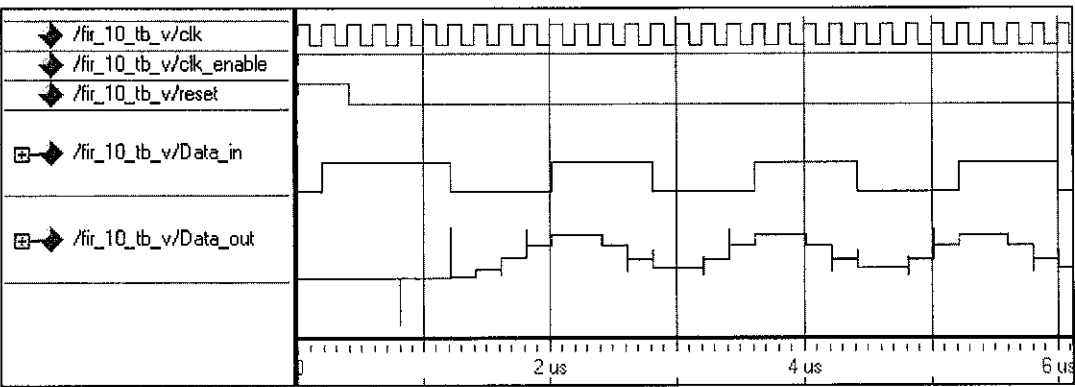


Figure 4.10: Timing simulation result for FIR filter

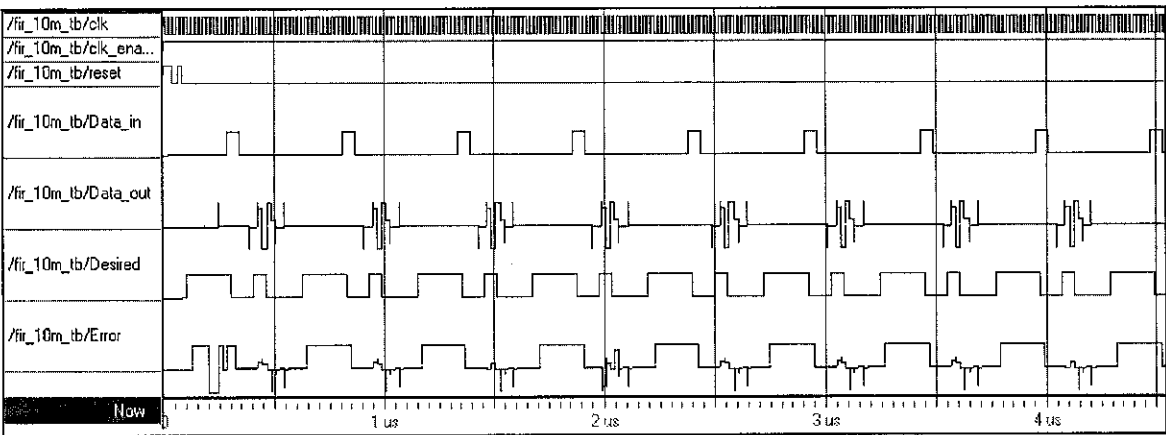


Figure 4.11: Timing simulation result for adaptive filter

The resulting simulation is displayed in the Waveform Viewer. The timing simulation result for FIR filter and adaptive filter looks similar with the result of behavioral simulation, however short timing delays caused by the hardware is introduced to the output. After simulations for the FIR filter and adaptive filter design obtain the desirable result, the generated bitstream is downloaded into the FPGA.

After supply the 10 MHz clock signal and input data to the FIR filter, the output of filter is measured by using logic analyzer. For this testing few set of data is load into the filter in FPGA. It is found that the output of filter from FPGA is same as the filter output from functional simulation. For example, the hexadecimal value ‘6000’ is loaded into the FIR filter, the same filter output from simulation and FPGA is ‘5fd0’.

CHAPTER 5

CONCLUSION AND RECOMMENDATIONS

5.1 Conclusion

The simulation on the adaptive noise cancellation system with Matlab, give the desired result as stated in the objective. The simulation results show that the noise can be cancelled by using the adaptive noise cancellation system, regardless of whether the input signal is a sine wave, music or a record of sound and the noise used is either white noise or chirp noise.

The result of filter is greatly depending on the order of filter. FIR filter with higher order will produce better result in Matlab simulation. The results indicate that the IIR filter does not necessary produce better result with higher order. As shown in table 4.1, the IIR filter with 25 orders introduces more noise to the output of adaptive filter compared with that of 10 orders.

From the Matlab simulation, it can be seen that LMS algorithm or NLMS algorithm with 10 orders IIR filter is suitable used for adaptive noise cancellation. However IIR filter cannot be used to implement adaptive filter in FPGA [15]. This is because the error at the IIR filter output is fed back to the input, hence error could be accumulated in the system.

The LMS algorithms decorrelates system output signal from the reference noise signal and removes noise components of the primary input signal based on second-order statistics only. However, there may be many other components in the primary input signal which depend on the noise reference signal through higher-order statistics.

After get the desired results for FIR filter and adaptive filter in the functional simulation and timing simulation, the filter design is downloaded into Virtex II FPGA device for hardware testing. However the logic analyzer used is not equipped with pattern generator function. Therefore the logic analyzer only can used to examine the output value instead of generate a sequence of test input to test the filter design. Hence the input data is supplied by connecting the input pins to 5 volt or ground to create an input data, while the clock is supplied from the function generator.

It is found that the FIR filter output from the FPGA is same as the simulation. That means the FIR filter is successfully implemented in FPGA. For the adaptive filter, the satisfied results is obtained for the Matlab simulation, functional simulation and timing simulation. Even though there is some error introduced to the filter design, such as quantization error, round off error and overflow error. The error not contributes to significant effect at the filter output, because the required precision of data is not as high as floating point precision.

In conclusion, the results of this project show that noise cancellation with adaptive filter is feasible. The noise is greatly reduced in Matlab simulation, functional simulation and timing simulation.

5.2 Recommendations

There are a few recommendations proposed to improve the performance of adaptive filter and to ease the filter design. The better results with Matlab simulation on adaptive noise cancellation system can be obtained by increasing the sampling frequency. So that the sound can be heard clearer and the difference between the filter output with different algorithm is more obvious.

Use with different types of FIR filter for the design, which including equiripple, window, least-square, constrained equiripple and so on. For more attenuation at the stopband, window and constrained equiripple is feasible for this purpose. Since the IIR filter is not suitable for the hardware implementation [15], the types of IIR filter is not recommended.

During the implementation of adaptive noise cancellation system, the recorded noise signal should be purely noise only without mixing with the desired input signal. Otherwise part of the origin signal will be cancelled at the output signal. Therefore it is important to determine the source of noise is not too close with the source of desired signal.

For those who interest to continue this project, low pass filter, high pass filter and bandpass filter can be implemented in the adaptive filter. This is to determine whether the algorithms used for the noise cancellation system can work well with different types of filters. Instead of doing with the Least-Mean-Square algorithm only, the noise cancellation system can be implemented with Normalised Least-Mean-Square algorithm, Recursive-Least-Squares algorithm or Kalman filter and Wiener filter.

Low power consumption and fast converging time are two essential factors need to be considered in implementing the noise cancellation system. It is necessary to find out which algorithms of adaptive filter consume less power and provide faster converging time for real time application. From the Matlab simulation, NLMS algorithm generate better result than the LMS algorithm, but it involve more

complex calculation, this may cause the converging time to be longer than LMS algorithm. For lower power consumption, instead of using direct form filter design, the transpose direct form filter should be used for the design [23]. The study on hardware algorithms for adder and multiplier should be carried out also to implement adaptive filter in hardware effectively.

Nowadays the software programming for the FPGA is not limited to the HDL languages only. A high-level design language such as C language is developed to ease the design flow of difficult algorithm application. Hardware designer can benefit from tools that allow them to mix high-level and low-level descriptions as needed to meet design goals as quickly as possible [21]. However it is still in early stages and is not yet a practical replacement for current HDL languages.

In general, a test bench is written to test the functionality of the design. Beside from this technique, the logic analyzer (option 3 from Agilent) with integrate pattern generator function can be used to generate the input data to test the functionality of the filter and adaptive noise cancellation system. The output can be display in the logic analyzer. In addition, the noise cancellation system can be designed with Xilinx System Generator. The testing on FPGA can be done by using co-simulation between System Generator and Matlab.

REFERENCES

1. M.G. Arnold, 1999. Verilog Digital Computer design, Algorithms to Hardware. Pearson Education, by Prentice Hall, Inc.
2. E. Lai, 2004. Practical Digital Signal Processing for Engineers and Technicians. Elsevier.
3. S. Hakyin, 2002. Adaptive Filter Theory. Prentice Hall, Inc.
4. "Example of adaptive noise cancellation", August 2005, <http://www.mathworks.com>.
5. A.Y. Lin & K.S. Gugel, 2003. "Feasibility of fixed-point transversal adaptive filters in FPGA devices with embedded DSP blocks". Applied Digital Design Laboratory, University of Florida
6. B. Jentz, 2005. "FPGAs rise to meet increasing DSP system requirements". Altera.
7. G. Hands, July 2004. High-performance DSP capability within an optimized FPGA, Lattice Semiconductor corporation
8. "Code: DSP", August 2005, [http:// www.altera.com/technology/dsp/dsp-code_dsp.html](http://www.altera.com/technology/dsp/dsp-code_dsp.html)
9. "FPGAs for DSP applications", August 2005, [http:// www.xilinx.com](http://www.xilinx.com)
10. Y. Tan, J. Wang & J.M. Zurada, 2001. "Nonlinear Blind Source Separation Using a Radial Basic Function Network". IEEE Transactions on Neural Networks, Vol. 12, No. 1.
11. H.M. Park, S.H. Oh, & S.Y. Lee, 2001. "On Adaptive Noise Cancelling Based on Independent Component Analysis". Korea Advanced Institute of Science and Technology.
12. M. Morris Mano, 2002. Digital Design, third edition. Prentice Hall
13. T.R. Padmanabhan, B. Bala Tripura Sundari, 2003. Design Through Verilog HDL. John Wiley & Sons, Inc.
14. Samir Palnitkar, 1996. Verilog HDL, A Guide to Digital Design and Synthesis. Sunsoft Press, A Prentice Hall Title.
15. Michael D. Ciletti, 2003. Advanced Digital Design with the Verilog HDL. Prentice Hall, Inc.
16. xilinc University Program, DSP Design Flow, Prefessor Workshop, 2003. Xilinx, Inc.

17. Virtex™-II Platform FPGAs: Complete Data Sheet. Xilinx. 2003.
18. Virtex-II XC2V1000 Reference Board User's Guide. Insight MEMEC, 2001.
19. "ISE 7.1i Quick Start Tutorial", October 2005, [http:// www.xilinx.com](http://www.xilinx.com)
20. J. Liang, R. Tessier, O. Mencer, "Floating point unit generation and evaluation for FPGAs," *Annual IEEE Symposium on Field - Programmable Custom Computing Machines*, 2003.
21. D. Pellerin, S. Thibault, 2005. Practical FPGA Programming in C. Prentice Hall
22. "DSP with FPGAs", August 2005, <http://www.andraka.com/dsp.htm>.
23. A. T. Erdogan and T. Arslan, 2000. "High throughput FIR filter design for low power SOC applications". Department of Electronics & Electrical Engineering, University of Edinburgh.

APPENDIX A

Gantt chart of final year project activities

A1. Planning activities for first semester

Planning Activities for First Semester of Final Year Project															
No.	Detail/ Week	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	Selection of Project Topic	■													
2	Preliminary Research on noise cancellation technique		■	■	■										
3	Submission of Preliminary Report			●											
4	Project Work				■	■	■	■							
	- Understand the Least-Mean-Square (LMS) algorithm				■	■	■	■							
	- Use matlab Simulink to design noise cancellation system				■	■	■	■							
	- Convert Simulink Diagram to matlab code				■	■	■	■							
5	Submission of Progress Report							●							
6	Project work continue								■	■	■	■			
	- Learnt the Verilog or VHDL program								■	■	■	■			
	- Write HDL code for a single filter								■	■	■	■			
	- Write test bench for a single filter								■	■	■	■			
7	Submission of Interim Report Final Draft											●			
8	Submission of Interim Report												●		
9	Oral Presentation														●

Table A.1: Gantt chart for activities of first semester

A2. Planning activities for second semester

Planning Activities for second Semester of Final Year Project																		
Detail/ Week	1	2	3	4	5	6	7	8	9	10	11	12	13	14	18	20		
Project Work Continue		■	■	■														
- Write HDL code for adaptive filter (LMS)		■	■	■														
- Write test bench for adaptive filter (LMS)		■	■	■														
Submission of Progress Report 1			●															
Project Work				■	■	■	■	■										
- Examines result of functional simulation for adaptive filter				■	■	■	■	■										
- Edit HDL code for adaptive filter (LMS)				■	■	■	■	■										
- Learnt to use ISE software				■	■	■	■	■										
Submission of Progress Report 2							●											
Project work continue								■	■	■	■	■	■					
- Examines result of timing simulation for adaptive filter								■	■	■	■	■	■					
- Edit HDL code for adaptive filter (LMS)								■	■	■	■	■	■					
- Write new HDL code into FPGA								■	■	■	■	■	■					
- Examine the adaptive filter with logic analyzer								■	■	■	■	■	■					
Submission of Dissertation Final Draft												●						
Submission of Final Report (soft cover)														●				
Submission of Technical Report															●			
Oral Presentation																●		
Submission of Project Dissertation (H.C)																	●	

Table A.2: Gantt chart for activities of second semester

APPENDIX B

Matlab code for sound recording and noise generation

To record sound in .wav format and save in c:

```
Fs = 8000;  
y = wavrecord(50*Fs, Fs, 'double');  
wavplay(y,Fs);  
wavwrite(y,Fs,'c:\music');
```

Generate the chirp signal in .wav file:

```
t=0:0.001:100;  
y=chirp(t,0,1,100,'q',[],'convex');  
fs=8000;  
wavwrite(y,fs,'chat');
```

APPENDIX C

Matlab code of noise cancellation with adaptive filter (LMS)

```
% Original signal
signal = sin(2*pi*0.055*(0:200-1)); %signal with f=0.055Hz, t:0 to 199
subplot(2,2,1),plot(0:199,signal(1:200));%plot signal with x: 0 to 199
grid; axis([0 200 -2 2]); %set the axis setting in the graph
title('The information of an original signal');

% The noise picked up by the secondary microphone is the input for the LMS
% adaptive filter.

nvar = 1.0; % Noise variance
noise = randn(200,1)*nvar; % White noise
subplot(2,2,2),plot(0:199,noise);
title('Noise picked up by the secondary microphone');
grid; axis([0 200 -4 4]);

% The noise corrupting the information a signal is a filtered version of 'noise':
% 31st order Low pass FIR filter, with Normalised f=0.5
nfilt = fir1(31,0.5);

% Filtering the noise
fnoise = filter(nfilt,1,noise);

% "Desired signal" for the adaptive filter (sine wave + filtered noise):
d=signal+fnoise;
subplot(2,2,3),plot(0:199,d(1:200));
grid; axis([0 200 -4 4]);
title('Desired input to the Adaptive Filter = Signal + Filtered Noise');

% adaptive filter with LMS algorithm

mu = 0.008; % LMS step size
Hadapt = adaptfilt.lms(32,mu); % filter order = 32
Hadapt.PersistentMemory = true; % for continuing updates the filter
weights
[y,e] = filter(Hadapt,noise,d);
H = abs(freqz(Hadapt,1,64));
H1 = abs(freqz(nfilt,1,64));

% Plot the frequency response of adaptive filter

wf = linspace(0,1,64);
plot(wf,H,wf,H1);
xlabel('Normalized Frequency (\times\pi rad/sample)');
ylabel('Magnitude');
legend('Adaptive Filter Response','Required Filter Response');
grid;
axis([0 1 0 2]);

% As the adaptive filter converges, the filtered noise should be completely
% subtracted from the "signal + noise" signal and the error signal 'e'
%should contains only the original signal.

subplot(2,2,4),plot(0:199,signal(1:200),0:199,e(1:200)); grid;
axis([0 200 -4 4]);
title('Original information of a signal and the error signal');
legend('Original Signal','Error Signal');
```

APPENDIX D

Verilog code for a 10 orders FIR filter

FIR filter

```
`timescale 1 ns / 1 ps

module fir_10(clk, clk_enable, reset, Data_in, Data_out);

    input    clk;
    input    clk_enable;
    input    reset;
    input    signed [15:0] Data_in;
    output   signed [15:0] Data_out;

    // FIR filter coefficient

    parameter signed [15:0] coeff1 = 16'b1111111100100010;
    parameter signed [15:0] coeff2 = 16'b1111111101101011;
    parameter signed [15:0] coeff3 = 16'b00000010001111100;
    parameter signed [15:0] coeff4 = 16'b00000111001010100;
    parameter signed [15:0] coeff5 = 16'b0001100100001010;
    parameter signed [15:0] coeff6 = 16'b0001101101111110;
    parameter signed [15:0] coeff7 = 16'b0001100100001010;
    parameter signed [15:0] coeff8 = 16'b00000111001010100;
    parameter signed [15:0] coeff9 = 16'b00000010001111100;
    parameter signed [15:0] coeff10 = 16'b1111111101101011;
    parameter signed [15:0] coeff11 = 16'b1111111100100010;

    // Declare vector used

    reg signed [15:0] delay [0:10] ;

    wire signed [31:0] product11, product10, product9, product8, product7;
    wire signed [31:0] product6, product5, product4, product3, product2, product1;

    wire signed [31:0] sum1, sum2, sum3, sum4, sum5;
    wire signed [31:0] sum6, sum7, sum8, sum9, sum10;

    wire signed [31:0] addsig, addsig_1, addsig_2, addsig_3;
    wire signed [31:0] addsig_4, addsig_5, addsig_6, addsig_7;
    wire signed [31:0] addsig_8, addsig_9, addsig_10, addsig_11;
    wire signed [31:0] addsig_12, addsig_13, addsig_14, addsig_15;
    wire signed [31:0] addsig_16, addsig_17, addsig_18, addsig_19;

    wire signed [32:0] add_temp, add_temp_1, add_temp_2, add_temp_3, add_temp_4;
    wire signed [32:0] add_temp_5, add_temp_6, add_temp_7, add_temp_8, add_temp_9;

    wire signed [15:0] output_round;
    reg signed [15:0] output_register;

    always @(posedge clk or posedge reset)

        // Reset, clear data

        begin: Delay_process
            if (reset == 1'b1) begin
                delay[0] <= 0;
                delay[1] <= 0;
                delay[2] <= 0;
                delay[3] <= 0;
                delay[4] <= 0;
                delay[5] <= 0;
                delay[6] <= 0;
                delay[7] <= 0;
                delay[8] <= 0;
                delay[9] <= 0;
                delay[10] <= 0;
            end
        end
```

```

// Transfer input data through all delay
else begin
    if (clk_enable == 1'b1) begin
        delay[0] <= Data_in;
        delay[1] <= delay[0];
        delay[2] <= delay[1];
        delay[3] <= delay[2];
        delay[4] <= delay[3];
        delay[5] <= delay[4];
        delay[6] <= delay[5];
        delay[7] <= delay[6];
        delay[8] <= delay[7];
        delay[9] <= delay[8];
        delay[10] <= delay[9];
    end
end
end // Delay_process

// Multiply the delay data with filter coefficients

assign product11 = delay[10] * coeff11;
assign product10 = delay[9] * coeff10;
assign product9 = delay[8] * coeff9;
assign product8 = delay[7] * coeff8;
assign product7 = delay[6] * coeff7;
assign product6 = delay[5] * coeff6;
assign product5 = delay[4] * coeff5;
assign product4 = delay[3] * coeff4;
assign product3 = delay[2] * coeff3;
assign product2 = delay[1] * coeff2;
assign product1 = delay[0] * coeff1;

// sum the result of multiplication

assign addsig = product1;
assign addsig_1 = product2;
assign add_temp = addsig + addsig_1;
assign sum1 = (add_temp[32] == 1'b0 & add_temp[31] != 1'b0) ?
32'b01111111111111111111111111111111 :
    (add_temp[32] == 1'b1 && add_temp[31] != 1'b1) ?
32'b10000000000000000000000000000000 : add_temp[31:0];

assign addsig_2 = sum1;
assign addsig_3 = product3;
assign add_temp_1 = addsig_2 + addsig_3;
assign sum2 = (add_temp_1[32] == 1'b0 & add_temp_1[31] != 1'b0) ?
32'b01111111111111111111111111111111 :
    (add_temp_1[32] == 1'b1 && add_temp_1[31] != 1'b1) ?
32'b10000000000000000000000000000000 : add_temp_1[31:0];

assign addsig_4 = sum2;
assign addsig_5 = product4;
assign add_temp_2 = addsig_4 + addsig_5;
assign sum3 = (add_temp_2[32] == 1'b0 & add_temp_2[31] != 1'b0) ?
32'b01111111111111111111111111111111 :
    (add_temp_2[32] == 1'b1 && add_temp_2[31] != 1'b1) ?
32'b10000000000000000000000000000000 : add_temp_2[31:0];

assign addsig_6 = sum3;
assign addsig_7 = product5;
assign add_temp_3 = addsig_6 + addsig_7;
assign sum4 = (add_temp_3[32] == 1'b0 & add_temp_3[31] != 1'b0) ?
32'b01111111111111111111111111111111 :
    (add_temp_3[32] == 1'b1 && add_temp_3[31] != 1'b1) ?
32'b10000000000000000000000000000000 : add_temp_3[31:0];

assign addsig_8 = sum4;
assign addsig_9 = product6;
assign add_temp_4 = addsig_8 + addsig_9;

```

```

    assign sum5 = (add_temp_4[32] == 1'b0 & add_temp_4[31] != 1'b0) ?
32'b01111111111111111111111111111111 :
    (add_temp_4[32] == 1'b1 && add_temp_4[31] != 1'b1) ?
32'b10000000000000000000000000000000 : add_temp_4[31:0];

    assign addsig_10 = sum5;
    assign addsig_11 = product7;
    assign add_temp_5 = addsig_10 + addsig_11;
    assign sum6 = (add_temp_5[32] == 1'b0 & add_temp_5[31] != 1'b0) ?
32'b01111111111111111111111111111111 :
    (add_temp_5[32] == 1'b1 && add_temp_5[31] != 1'b1) ?
32'b10000000000000000000000000000000 : add_temp_5[31:0];

    assign addsig_12 = sum6;
    assign addsig_13 = product8;
    assign add_temp_6 = addsig_12 + addsig_13;
    assign sum7 = (add_temp_6[32] == 1'b0 & add_temp_6[31] != 1'b0) ?
32'b01111111111111111111111111111111 :
    (add_temp_6[32] == 1'b1 && add_temp_6[31] != 1'b1) ?
32'b10000000000000000000000000000000 : add_temp_6[31:0];

    assign addsig_14 = sum7;
    assign addsig_15 = product9;
    assign add_temp_7 = addsig_14 + addsig_15;
    assign sum8 = (add_temp_7[32] == 1'b0 & add_temp_7[31] != 1'b0) ?
32'b01111111111111111111111111111111 :
    (add_temp_7[32] == 1'b1 && add_temp_7[31] != 1'b1) ?
32'b10000000000000000000000000000000 : add_temp_7[31:0];

    assign addsig_16 = sum8;
    assign addsig_17 = product10;
    assign add_temp_8 = addsig_16 + addsig_17;
    assign sum9 = (add_temp_8[32] == 1'b0 & add_temp_8[31] != 1'b0) ?
32'b01111111111111111111111111111111 :
    (add_temp_8[32] == 1'b1 && add_temp_8[31] != 1'b1) ?
32'b10000000000000000000000000000000 : add_temp_8[31:0];

    assign addsig_18 = sum9;
    assign addsig_19 = product11;
    assign add_temp_9 = addsig_18 + addsig_19;
    assign sum10 = (add_temp_9[32] == 1'b0 & add_temp_9[31] != 1'b0) ?
32'b01111111111111111111111111111111 :
    (add_temp_9[32] == 1'b1 && add_temp_9[31] != 1'b1) ?
32'b10000000000000000000000000000000 : add_temp_9[31:0];

    assign output_round = (sum10[31] == 1'b0 & sum10[30:29] != 2'b00) ?
16'b0111111111111111 :
    (sum10[31] == 1'b1 && sum10[30:29] != 2'b11) ? 16'b1000000000000000 :
sum10[29:14];

    always @ (posedge clk or posedge reset)
    begin: Output_Register_process
        if (reset == 1'b1) begin
            output_register <= 0;
        end
        else begin
            if (clk_enable == 1'b1) begin
                output_register <= output_round;
            end
        end
    end // Output_Register_process

    // Assignment Statements
    assign Data_out = output_register;

endmodule // fir_10

```


Testbench of 10 orders FIR filter

```
`timescale 1 ns / 1 ps

module fir_10_tb_v;

    // Parameters
    parameter clk_high   = 100;
    parameter clk_low    = 100;
    parameter clk_period = 200;
    parameter clk_hold   = 10;

    // Inputs
    reg clk;
    reg clk_enable;
    reg reset;
    reg signed [15:0] Data_in;

    // Outputs
    wire signed [15:0] Data_out;

    integer n; //loop variable

    reg signed [15:0] Data_in_load[0:79];
    reg signed [15:0] Data_out_expected
    [0:79];

    // Instantiate the Unit Under
    Test (UUT)
        fir_10 uut (
            .clk(clk),
            .clk_enable(clk_enable),
            .reset(reset),
            .Data_in(Data_in),
            .Data_out(Data_out)
        );

    initial
        begin
            // Constants
            Data_in_load[0] <= 16'h4000;
            Data_in_load[1] <= 16'h4000;
            Data_in_load[2] <= 16'h4000;
            Data_in_load[3] <= 16'h4000;
            Data_in_load[4] <= 16'h0000;
            Data_in_load[5] <= 16'h0000;
            Data_in_load[6] <= 16'h0000;
            Data_in_load[7] <= 16'h0000;
            Data_in_load[8] <= 16'h4000;
            Data_in_load[9] <= 16'h4000;
            Data_in_load[10] <= 16'h4000;
            Data_in_load[11] <= 16'h4000;
            Data_in_load[12] <= 16'h0000;
            Data_in_load[13] <= 16'h0000;
            Data_in_load[14] <= 16'h0000;
            Data_in_load[15] <= 16'h0000;
            Data_in_load[16] <= 16'h4000;
            Data_in_load[17] <= 16'h4000;
            Data_in_load[18] <= 16'h4000;
            Data_in_load[19] <= 16'h4000;
            Data_in_load[20] <= 16'h0000;
            Data_in_load[21] <= 16'h0000;
            Data_in_load[22] <= 16'h0000;
            Data_in_load[23] <= 16'h0000;
            Data_in_load[24] <= 16'h4000;
            Data_in_load[25] <= 16'h4000;
            Data_in_load[26] <= 16'h4000;
            Data_in_load[27] <= 16'h4000;
            Data_in_load[28] <= 16'h0000;
            Data_in_load[29] <= 16'h0000;
            Data_in_load[30] <= 16'h0000;

            Data_in_load[31] <= 16'h0000;
            Data_in_load[32] <= 16'h4000;
            Data_in_load[33] <= 16'h4000;
            Data_in_load[34] <= 16'h4000;
            Data_in_load[35] <= 16'h4000;
            Data_in_load[36] <= 16'h0000;
            Data_in_load[37] <= 16'h0000;
            Data_in_load[38] <= 16'h0000;
            Data_in_load[39] <= 16'h0000;
            Data_in_load[40] <= 16'h4000;
            Data_in_load[41] <= 16'h4000;
            Data_in_load[42] <= 16'h4000;
            Data_in_load[43] <= 16'h4000;
            Data_in_load[44] <= 16'h0000;
            Data_in_load[45] <= 16'h0000;
            Data_in_load[46] <= 16'h0000;
            Data_in_load[47] <= 16'h0000;
            Data_in_load[48] <= 16'h4000;
            Data_in_load[49] <= 16'h4000;
            Data_in_load[50] <= 16'h4000;
            Data_in_load[51] <= 16'h4000;
            Data_in_load[52] <= 16'h0000;
            Data_in_load[53] <= 16'h0000;
            Data_in_load[54] <= 16'h0000;
            Data_in_load[55] <= 16'h0000;
            Data_in_load[56] <= 16'h4000;
            Data_in_load[57] <= 16'h4000;
            Data_in_load[58] <= 16'h4000;
            Data_in_load[59] <= 16'h4000;
            Data_in_load[60] <= 16'h0000;
            Data_in_load[61] <= 16'h0000;
            Data_in_load[62] <= 16'h0000;
            Data_in_load[63] <= 16'h0000;
            Data_in_load[64] <= 16'h4000;
            Data_in_load[65] <= 16'h4000;
            Data_in_load[66] <= 16'h4000;
            Data_in_load[67] <= 16'h4000;
            Data_in_load[68] <= 16'h0000;
            Data_in_load[69] <= 16'h0000;
            Data_in_load[70] <= 16'h0000;
            Data_in_load[71] <= 16'h0000;
            Data_in_load[72] <= 16'h4000;
            Data_in_load[73] <= 16'h4000;
            Data_in_load[74] <= 16'h4000;
            Data_in_load[75] <= 16'h4000;
            Data_in_load[76] <= 16'h0000;
            Data_in_load[77] <= 16'h0000;
            Data_in_load[78] <= 16'h0000;
            Data_in_load[79] <= 16'h0000;

            Data_out_expected [0] <= 16'hff22;
            Data_out_expected [1] <= 16'hfed9;
            Data_out_expected [2] <= 16'h0355;
            Data_out_expected [3] <= 16'h11a9;
            Data_out_expected [4] <= 16'h2b91;
            Data_out_expected [5] <= 16'h4998;
            Data_out_expected [6] <= 16'h5e26;
            Data_out_expected [7] <= 16'h5e26;
            Data_out_expected [8] <= 16'h48ba;
            Data_out_expected [9] <= 16'h2a6a;
            Data_out_expected [10] <= 16'h14fe;
            Data_out_expected [11] <= 16'h14fe;
            Data_out_expected [12] <= 16'h2a6a;
            Data_out_expected [13] <= 16'h48ba;
            Data_out_expected [14] <= 16'h5e26;
            Data_out_expected [15] <= 16'h5e26;
            Data_out_expected [16] <= 16'h48ba;
            Data_out_expected [17] <= 16'h2a6a;
            Data_out_expected [18] <= 16'h14fe;
            Data_out_expected [19] <= 16'h14fe;
            Data_out_expected [20] <= 16'h2a6a;
```

```

Data_out_expected [21] <= 16'h48ba;
Data_out_expected [22] <= 16'h5e26;
Data_out_expected [23] <= 16'h5e26;
Data_out_expected [24] <= 16'h48ba;
Data_out_expected [25] <= 16'h2a6a;
Data_out_expected [26] <= 16'h14fe;
Data_out_expected [27] <= 16'h14fe;
Data_out_expected [28] <= 16'h2a6a;
Data_out_expected [29] <= 16'h48ba;
Data_out_expected [30] <= 16'h5e26;
Data_out_expected [31] <= 16'h5e26;
Data_out_expected [32] <= 16'h48ba;
Data_out_expected [33] <= 16'h2a6a;
Data_out_expected [34] <= 16'h14fe;
Data_out_expected [35] <= 16'h14fe;
Data_out_expected [36] <= 16'h2a6a;
Data_out_expected [37] <= 16'h48ba;
Data_out_expected [38] <= 16'h5e26;
Data_out_expected [39] <= 16'h5e26;
Data_out_expected [40] <= 16'h48ba;
Data_out_expected [41] <= 16'h2a6a;
Data_out_expected [42] <= 16'h14fe;
Data_out_expected [43] <= 16'h14fe;
Data_out_expected [44] <= 16'h2a6a;
Data_out_expected [45] <= 16'h48ba;
Data_out_expected [46] <= 16'h5e26;
Data_out_expected [47] <= 16'h5e26;
Data_out_expected [48] <= 16'h48ba;
Data_out_expected [49] <= 16'h2a6a;
Data_out_expected [50] <= 16'h14fe;
Data_out_expected [51] <= 16'h14fe;
Data_out_expected [52] <= 16'h2a6a;
Data_out_expected [53] <= 16'h48ba;
Data_out_expected [54] <= 16'h5e26;
Data_out_expected [55] <= 16'h5e26;
Data_out_expected [56] <= 16'h48ba;
Data_out_expected [57] <= 16'h2a6a;
Data_out_expected [58] <= 16'h14fe;
Data_out_expected [59] <= 16'h14fe;
Data_out_expected [60] <= 16'h2a6a;
Data_out_expected [61] <= 16'h48ba;
Data_out_expected [62] <= 16'h5e26;
Data_out_expected [63] <= 16'h5e26;
Data_out_expected [64] <= 16'h48ba;
Data_out_expected [65] <= 16'h2a6a;
Data_out_expected [66] <= 16'h14fe;
Data_out_expected [67] <= 16'h14fe;
Data_out_expected [68] <= 16'h2a6a;
Data_out_expected [69] <= 16'h48ba;
Data_out_expected [70] <= 16'h5e26;
Data_out_expected [71] <= 16'h5e26;
Data_out_expected [72] <= 16'h48ba;
Data_out_expected [73] <= 16'h2a6a;

Data_out_expected [74] <= 16'h14fe;
Data_out_expected [75] <= 16'h14fe;
Data_out_expected [76] <= 16'h2a6a;
Data_out_expected [77] <= 16'h48ba;
Data_out_expected [78] <= 16'h5e26;
Data_out_expected [79] <= 16'h5e26;

end

// Block Statements
always // clk generation
begin : clk_gen
    clk <= 1'b 1;
    # clk_high;
    clk <= 1'b 0;
    # clk_low;
end //clk_gen;

initial // reset block
begin : reset_gen
    clk_enable <= 1'b1;
    reset <= 1'b 1;
    # (clk_period*2 + clk_hold);
    reset <= 1'b 0;
end //reset_gen;

initial //The main block
begin
    # clk_period;
    Data_in <= Data_in_load[0];
    # (clk_period*2 + clk_hold);
    Data_in <= Data_in_load[1];
    # clk_period;
    for (n = 0; n <= 79; n = n + 1)
        begin
            if (Data_out !=
Data_out_expected[n])
                $display("ERROR in filter test at
time %t : Expected '%h' Actual '%h'",
$time, Data_out_expected[n], Data_out);
            if (n + 2 <= 79)
                Data_in <= Data_in_load[n + 2];
            # (clk_period);
        end
        $display( "**** Test Complete. ****"
);
        $stop;
    end //Data_in_gen;

endmodule

```

APPENDIX E

Verilog code for a 10 order adaptive filter

Adaptive filter

```
`timescale 1 ns / 1 ps

module fir_10m (clk, clk_enable, reset, Data_in, Data_out, Desired,
Error,mult,temp,mumu,mumus);

    input    clk;
    input    clk_enable;
    input    reset;
    input    signed [15:0] Data_in;
    output   signed [15:0] Data_out;

    input    signed [15:0] Desired;
    output   signed [15:0] Error;
    output   signed [31:0] mult;
    output   signed [15:0] temp;
    output   signed [31:0] mumu;
    output   signed [15:0] mumus;

    reg    signed [15:0] c[0:10]; //weight
    reg    signed [15:0] w[0:10];
    reg    signed [31:0] mult1;
    reg    signed [15:0] temp1;
    reg    signed [31:0] mumu1;
    reg    signed [15:0] mumu2;

    wire    signed [15:0] w0,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10;
    wire    signed [15:0] ce0,ce1,ce2,ce3,ce4,ce5,ce6,ce7,ce8,ce9,ce10;

    //parameter    mu=0.05 (16'h0666);
    reg    signed [15:0] mu = 16'h0166;

    reg    signed [15:0] delay [0:10] ;

    wire signed [31:0] product11, product10, product9, product8, product7;
    wire signed [31:0] product6, product5, product4, product3, product2, product1;

    wire signed [31:0] sum1, sum2, sum3, sum4, sum5;
    wire signed [31:0] sum6, sum7, sum8, sum9, sum10;

    wire signed [31:0] addsig, addsig_1, addsig_2,addsig_3;
    wire signed [31:0] addsig_4, addsig_5, addsig_6,addsig_7;
    wire signed [31:0] addsig_8, addsig_9, addsig_10,addsig_11;
    wire signed [31:0] addsig_12, addsig_13, addsig_14,addsig_15;
    wire signed [31:0] addsig_16, addsig_17, addsig_18,addsig_19;

    wire signed [32:0] add_temp, add_temp_1, add_temp_2, add_temp_3, add_temp_4;
    wire signed [32:0] add_temp_5, add_temp_6, add_temp_7, add_temp_8, add_temp_9;

    wire signed [15:0] output_round; // sfix16_En15
    reg    signed [15:0] output_register; // sfix16_En15
    reg    signed [15:0] error_m;

    // Load input data
    always @( posedge clk or posedge reset)
        begin: Delay_process
            if (reset == 1'b1) begin
                delay[0] <= 0;
                delay[1] <= 0;
                delay[2] <= 0;
                delay[3] <= 0;
                delay[4] <= 0;
                delay[5] <= 0;
                delay[6] <= 0;
            end
        end
    endmodule
```

```

        delay[7] <= 0;
        delay[8] <= 0;
        delay[9] <= 0;
        delay[10] <= 0;
    end
    else begin
        if (clk_enable == 1'b1) begin
            delay[0] <= Data_in;
            delay[1] <= delay[0];
            delay[2] <= delay[1];
            delay[3] <= delay[2];
            delay[4] <= delay[3];
            delay[5] <= delay[4];
            delay[6] <= delay[5];
            delay[7] <= delay[6];
            delay[8] <= delay[7];
            delay[9] <= delay[8];
            delay[10] <= delay[9];
            //$display ("delay[0]:%h, delay[1]:%h",delay[0],delay[1]);
        end
    end
end // Delay_process

//Load coefficients
always @(posedge clk)
begin
    if (reset==1'b1)begin
        c[0] <= 16'h0000;//16'h0187;
        c[1] <= 16'h0000;//16'hfdc4;
        c[2] <= 16'hfd46;//16'h0361;
        c[3] <= 16'h0000;//16'hfb90;
        c[4] <= 16'h229a;//16'h0530;
        c[5] <= 16'h4000;//16'h7a8b;
        c[6] <= 16'h229a;//16'h0530;
        c[7] <= 16'h0000;//16'hfb90;
        c[8] <= 16'hfd46;//16'h0361;
        c[9] <= 16'h0000;//16'hfdc4;
        c[10] <= 16'h0000;//16'h0187;
    end
end

//multiply input data with coefficients
assign product11 = delay[10] * c[10];
assign product10 = delay[9] * c[9];
assign product9 = delay[8] * c[8];
assign product8 = delay[7] * c[7];
assign product7 = delay[6] * c[6];
assign product6 = delay[5] * c[5];
assign product5 = delay[4] * c[4];
assign product4 = delay[3] * c[3];
assign product3 = delay[2] * c[2];
assign product2 = delay[1] * c[1];
assign product1 = delay[0] * c[0];

//Sum all result of multiplication
assign addsig = product1;
assign addsig_1 = product2;
assign add_temp = addsig + addsig_1;
assign sum1 = (add_temp[32:31] == 2'b01) ? 32'h7fffffff :
    (add_temp[32:31] == 2'b10) ? 32'h80000000 : add_temp[31:0];

assign addsig_2 = sum1;
assign addsig_3 = product3;
assign add_temp_1 = addsig_2 + addsig_3;
assign sum2 = (add_temp_1[32:31] == 2'b01) ? 32'h7fffffff :
    (add_temp_1[32:31] == 2'b10) ? 32'h80000000 : add_temp_1[31:0];

assign addsig_4 = sum2;
assign addsig_5 = product4;
assign add_temp_2 = addsig_4 + addsig_5;
assign sum3 = (add_temp_2[32:31] == 2'b01) ? 32'h7fffffff :
    (add_temp_2[32:31] == 2'b10) ? 32'h80000000 : add_temp_2[31:0];

```

```

assign addsig_6 = sum3;
assign addsig_7 = product5;
assign add_temp_3 = addsig_6 + addsig_7;
assign sum4 = (add_temp_3[32:31] == 2'b01) ? 32'h7fffffff :
    (add_temp_3[32:31] == 2'b10) ? 32'h80000000 : add_temp_3[31:0];

assign addsig_8 = sum4;
assign addsig_9 = product6;
assign add_temp_4 = addsig_8 + addsig_9;
assign sum5 = (add_temp_4[32:31] == 2'b01) ? 32'h7fffffff :
    (add_temp_4[32:31] == 2'b10) ? 32'h80000000 : add_temp_4[31:0];

assign addsig_10 = sum5;
assign addsig_11 = product7;
assign add_temp_5 = addsig_10 + addsig_11;
assign sum6 = (add_temp_5[32:31] == 2'b01) ? 32'h7fffffff :
    (add_temp_5[32:31] == 2'b10) ? 32'h80000000 : add_temp_5[31:0];

assign addsig_12 = sum6;
assign addsig_13 = product8;
assign add_temp_6 = addsig_12 + addsig_13;
assign sum7 = (add_temp_6[32:31] == 2'b01) ? 32'h7fffffff :
    (add_temp_6[32:31] == 2'b10) ? 32'h80000000 : add_temp_6[31:0];

assign addsig_14 = sum7;
assign addsig_15 = product9;
assign add_temp_7 = addsig_14 + addsig_15;
assign sum8 = (add_temp_7[32:31] == 2'b01) ? 32'h7fffffff :
    (add_temp_7[32:31] == 2'b10) ? 32'h80000000 : add_temp_7[31:0];

assign addsig_16 = sum8;
assign addsig_17 = product10;
assign add_temp_8 = addsig_16 + addsig_17;
assign sum9 = (add_temp_8[32:31] == 2'b01) ? 32'h7fffffff :
    (add_temp_8[32:31] == 2'b10) ? 32'h80000000 : add_temp_8[31:0];

assign addsig_18 = sum9;
assign addsig_19 = product11;
assign add_temp_9 = addsig_18 + addsig_19;
assign sum10 = (add_temp_9[32:31] == 2'b01) ? 32'h7fffffff :
    (add_temp_9[32:31] == 2'b10) ? 32'h80000000 : add_temp_9[31:0];

//take 16 bits data from the sum
assign output_round = (sum10[31] == 1'b0 & sum10[30] != 1'b0) ? 16'h7fff :
    (sum10[31] == 1'b1 && sum10[30] != 1'b1) ? 16'h8000 : sum10[30:15];

always @ (posedge clk or posedge reset)
begin: Output_Register_process

    if (reset == 1'b1) begin
        output_register <= 0;
    end

    else begin
        if (clk_enable == 1'b1) begin
            output_register = output_round;
            error_m = Desired-output_register;

            //c[n] <= c[n]+mu*Data_in*error_m;

            mult1 = error_m*Data_in;
            temp1 = mult1[31:16];
            mumu1 = mu*temp1;
            mumu2 = mumu1[31:16];

            //
            for (n=0; n<11; n=n+1)begin
            //
                c[n]=c[n]+mumu2;
            //
            end

            c[0] = c[0]+mumu2;
            c[1] = c[1]+mumu2;
            c[2] = c[2]+mumu2;
            c[3] = c[3]+mumu2;
            c[4] = c[4]+mumu2;

```

```

        c[5]  = c[5]+mumu2;
        c[6]  = c[6]+mumu2;
        c[7]  = c[7]+mumu2;
        c[8]  = c[8]+mumu2;
        c[9]  = c[9]+mumu2;
        c[10] = c[10]+mumu2;

        end //clk_enable
    end //else
end // Output_Register_process
// Assignment Statements

assign Data_out = output_register;
assign Error = error_m;
assign mult = mult1;
assign temp = temp1;
assign mumu = mumu1;
assign mumu2 = mumu2;

assign ce0 = c[0];
assign ce1 = c[1];
assign ce2 = c[2];
assign ce3 = c[3];
assign ce4 = c[4];
assign ce5 = c[5];
assign ce6 = c[6];
assign ce7 = c[7];
assign ce8 = c[8];
assign ce9 = c[9];
assign ce10 = c[10];

assign w0 = delay[0];
assign w1 = delay[1];
assign w2 = delay[2];
assign w3 = delay[3];
assign w4 = delay[4];
assign w5 = delay[5];
assign w6 = delay[6];
assign w7 = delay[7];
assign w8 = delay[8];
assign w9 = delay[9];
assign w10 = delay[10];

endmodule // fir_10m

```

Testbench of adaptive filter

```
`timescale 1 ns / 1 ps

module fir_10m_tb;

    // Parameters
    parameter clk_high   = 10;
    parameter clk_low    = 10;
    parameter clk_period = 20;
    parameter clk_hold   = 4;

    // Nets
    reg clk;
    reg clk_enable;
    reg reset;
    reg signed [15:0] Data_in;
    wire signed [15:0] Data_out;

    reg signed [15:0] Desired;
    wire signed [15:0] Error;
    wire signed [31:0] mult;
    wire signed [15:0] temp;
    wire signed [31:0] mumu;
    wire signed [15:0] mumus;

    integer n; //loop variable
    integer k;

    reg signed [15:0] Data_in_load[0:21];
    reg signed [15:0] Data_out_expected [0:21];

    reg signed [15:0] Desired_data [0:21];

    // Component Instances
    fir_10m u_fir_10m
    (
        .clk(clk),
        .clk_enable(clk_enable),
        .reset(reset),
        .Data_in(Data_in),
        .Data_out(Data_out),
        .Desired(Desired),
        .Error(Error),
        .mult(mult),
        .temp(temp),
        .mumu(mumu),
        .mumus(mumus)
    );

    initial
    begin
        Data_out_expected [0] <= 16'h0000;
        Data_out_expected [1] <= 16'h0000;
        Data_out_expected [2] <= 16'h0000;
        Data_out_expected [3] <= 16'hfd46;
        Data_out_expected [4] <= 16'hfd46;
        Data_out_expected [5] <= 16'h1fdf;
        Data_out_expected [6] <= 16'h5fdf;
        Data_out_expected [7] <= 16'h7fff;
        Data_out_expected [8] <= 16'h7fff;
        Data_out_expected [9] <= 16'h7fbf;
        Data_out_expected [10] <= 16'h7fbf;
        Data_out_expected [11] <= 16'h7fbf;
        Data_out_expected [12] <= 16'h7fbf;
        Data_out_expected [13] <= 16'h7fff;
        Data_out_expected [14] <= 16'h7fff;
        Data_out_expected [15] <= 16'h5fdf;
        Data_out_expected [16] <= 16'h1fdf;
        Data_out_expected [17] <= 16'hfd46;
        Data_out_expected [18] <= 16'hfd46;
        Data_out_expected [19] <= 16'h0000;
        Data_out_expected [20] <= 16'h0000;
        Data_out_expected [21] <= 16'h0000;
    end
endmodule
```

```

// Reference input u(n) as an impulse noise
Data_in_load[0] <= 16'h0050;
Data_in_load[1] <= 16'h0050;
Data_in_load[2] <= 16'h0050;
Data_in_load[3] <= 16'h0050;
Data_in_load[4] <= 16'h0050;
Data_in_load[5] <= 16'h0050;
Data_in_load[6] <= 16'h0050;
Data_in_load[7] <= 16'h0050;
Data_in_load[8] <= 16'h0050;
Data_in_load[9] <= 16'h0050;
Data_in_load[10] <= 16'h0050;
Data_in_load[11] <= 16'h0050;
Data_in_load[12] <= 16'h7fff;
Data_in_load[13] <= 16'h7fff;
Data_in_load[14] <= 16'h7fff;
Data_in_load[15] <= 16'h0050;
Data_in_load[16] <= 16'h0050;
Data_in_load[17] <= 16'h0050;
Data_in_load[18] <= 16'h0050;
Data_in_load[19] <= 16'h0050;
Data_in_load[20] <= 16'h0050;
Data_in_load[21] <= 16'h0050;

//Primary input d(n), which is input(step signal) + noise
Desired_data [0] <= 16'h0050;
Desired_data [1] <= 16'h7fff;
Desired_data [2] <= 16'h7fff;
Desired_data [3] <= 16'h7fff;
Desired_data [4] <= 16'h7fff;
Desired_data [5] <= 16'h7fff;
Desired_data [6] <= 16'h7fff;
Desired_data [7] <= 16'h7fff;
Desired_data [8] <= 16'h7fff;
Desired_data [9] <= 16'h7fff;
Desired_data [10] <= 16'h7fff;
Desired_data [11] <= 16'h0050;
Desired_data [12] <= 16'h0050;
Desired_data [13] <= 16'h0050;
Desired_data [14] <= 16'h0050;
Desired_data [15] <= 16'h0050;
Desired_data [16] <= 16'h7fff;
Desired_data [17] <= 16'h7fff;
Desired_data [18] <= 16'h7fff;
Desired_data [19] <= 16'h0050;
Desired_data [20] <= 16'h0050;
Desired_data [21] <= 16'h0050;
end

always // clk generation
begin : clk_gen
clk <= 1'b 1;
# clk_high;
clk <= 1'b 0;
# clk_low;
end //clk_gen;

initial // reset block
begin : reset_gen
clk_enable <= 1'b1;
reset <= 1'b 1;
# (clk_period*2 + clk_hold);
reset <= 1'b 0;
# 20;
reset <= 1'b1;
# 20;
reset <= 1'b0;
end //reset_gen;

```



```

initial //The main block
begin
  for (k=0;k<1000;k=k+1) begin
    # clk_period;
    Data_in <= Data_in_load[0];
    # (clk_period*2 + clk_hold);
    Data_in <= Data_in_load[1];

    # clk_period;
    for (n = 0; n<= 21; n = n + 1)
      begin
        if ((n + 2) <= 21)
          Data_in <= Data_in_load[n + 2];
          $display("%5d=> Data_in: %5h ; error: %5h ; mult: %5h ; temp: %5h ; mumu:
%5h ; mumus: %5h", n, Data_in,Error,mult,temp,mumu,mumus);
          Desired <= Desired_data[n];
          # (clk_period);

        end
      end
    $display( "**** Test Complete. ****" );
    $stop;

  end //Data_in_gen;

endmodule

```

APPENDIX F

Performance of Virtex-II

Table 11: Pin-to-Pin Performance

Description	Device Used & Speed Grade	Pin-to-Pin (with I/O delays)	Units
Basic Functions			
16-bit Address Decoder	XC2V1000 -5	6.3	ns
32-bit Address Decoder	XC2V1000 -5	7.7	ns
64-bit Address Decoder	XC2V1000 -5	9.3	ns
4:1 MUX	XC2V1000 -5	5.7	ns
8:1 MUX	XC2V1000 -5	6.5	ns
16:1 MUX	XC2V1000 -5	6.7	ns
32:1 MUX	XC2V1000 -5	8.7	ns
Combinatorial (pad to LUT to pad)	XC2V1000 -5	5.0	ns
Memory			
Block RAM			
Pad to setup		1.6	ns
Clock to Pad		9.5	ns
Distributed RAM			
Pad to setup	XC2V1000 -5	2.7	ns
Clock to Pad	XC2V1000 -5	5.1 (no clk skew)	ns

Table 12 shows internal (register-to-register) performance. Values are reported in MHz.

Table 12: Register-to-Register Performance

Description	Device Used & Speed Grade	Register-to-Register Performance	Units
Basic Functions			
16-bit Address Decoder	XC2V1000 -5	398	MHz
32-bit Address Decoder	XC2V1000 -5	291	MHz
64-bit Address Decoder	XC2V1000 -5	274	MHz
4:1 MUX	XC2V1000 -5	563	MHz
8:1 MUX	XC2V1000 -5	454	MHz
16:1 MUX	XC2V1000 -5	414	MHz
32:1 MUX	XC2V1000 -5	323	MHz
Register to LUT to Register	XC2V1000 -5	613	MHz

Table 12: Register-to-Register Performance (Continued)

Description	Device Used & Speed Grade	Register-to-Register Performance	Units
8-bit Adder	XC2V1000 -5	292	MHz
16-bit Adder	XC2V1000 -5	239	MHz
64-bit Adder	XC2V1000 -5	114	MHz
64-bit Counter	XC2V1000 -5	114	MHz
64-bit Accumulator	XC2V1000 -5	110	MHz
Multiplier 18x18 (with Block RAM inputs)	XC2V1000 -5	98	MHz
Multiplier 18x18 (with Register inputs)	XC2V1000 -5	105	MHz
Memory			
Block RAM			
Single-Port 4096 x 4 bits		278	MHz
Single-Port 2048 x 9 bits		277	MHz
Single-Port 1024 x 18 bits		270	MHz
Single-Port 512 x 36 bits		259	MHz
Dual-Port A: 4096 x 4 bits & B: 1024 x 18 bits		257	MHz
Dual-Port A: 1024 x 18 bits & B: 1024 x 18 bits		259	MHz
Dual-Port A: 2048 x 9 bits & B: 512 x 36 bits		250	MHz
Distributed RAM			
Single-Port 32 x 8-bit	XC2V1000 -5	997	MHz
Single-Port 64 x 8-bit	XC2V1000 -5	935	MHz
Single-Port 128 x 8-bit	XC2V1000 -5	866	MHz
Dual-Port 16 x 8	XC2V1000 -5	409	MHz
Dual-Port 32 x 8	XC2V1000 -5	911	MHz
Dual-Port 64 x 8	XC2V1000 -5	294	MHz
Shift Registers			
128-bit SRL		N/A	MHz
256-bit SRL		N/A	MHz
FIFOs (Async. In Block RAM)			
1024 x 18-bit Read		279	MHz
1024 x 18-bit Write		172	MHz
FIFOs (Sync. In SRL)			
128 x 8-bit		N/A	MHz
128 x 16-bit		N/A	MHz

APPENDIX G

Simulation result of adaptive noise cancellation system with simulink

FIR filter (order: 25) with NLMS

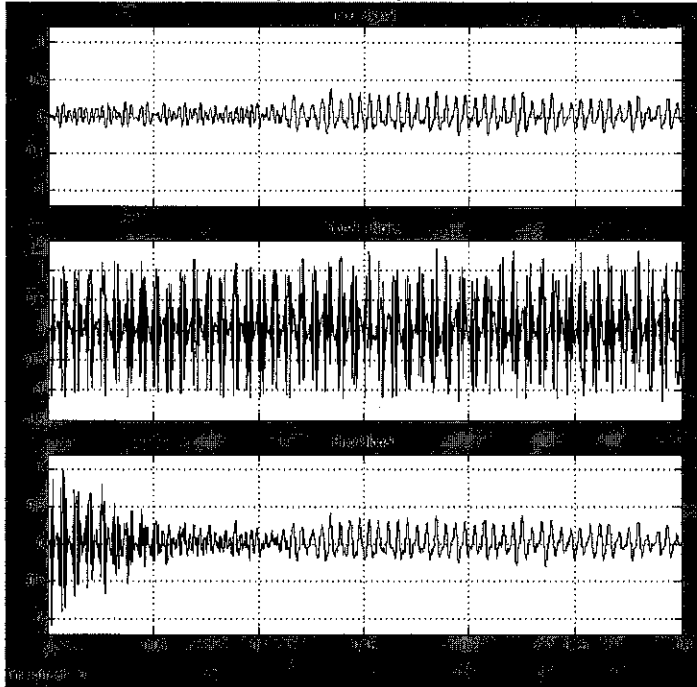


Figure G.1: Input signal, interfered signal and output signal of adaptive noise cancellation system with NLMS algorithm and 25 orders FIR filter.

IIR filter (order: 25) with NLMS

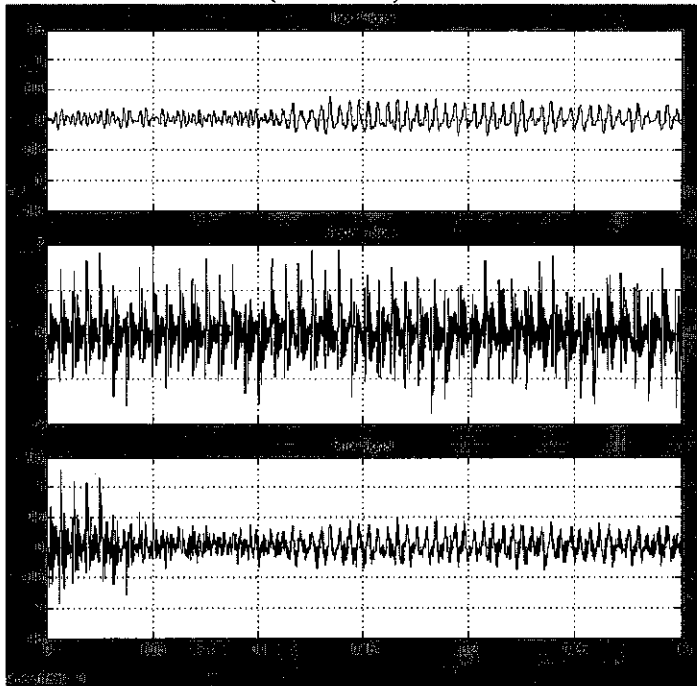


Figure G.2: Input signal, interfered signal and output signal of adaptive noise cancellation system with NLMS algorithm and 25 orders IIR filter.

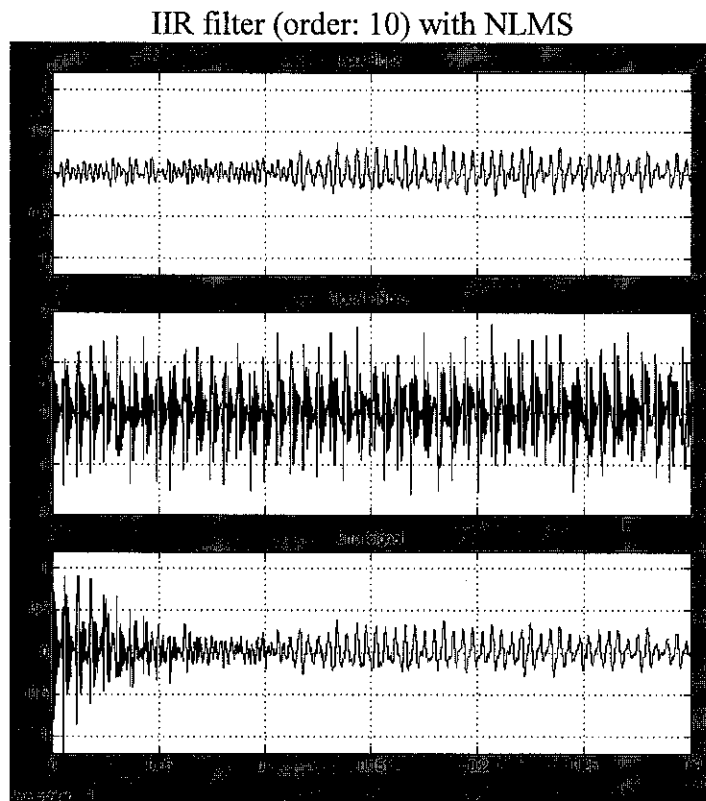


Figure G.3: Input signal, interfered signal and output signal of adaptive noise cancellation system with NLMS algorithm and 10 orders IIR filter.

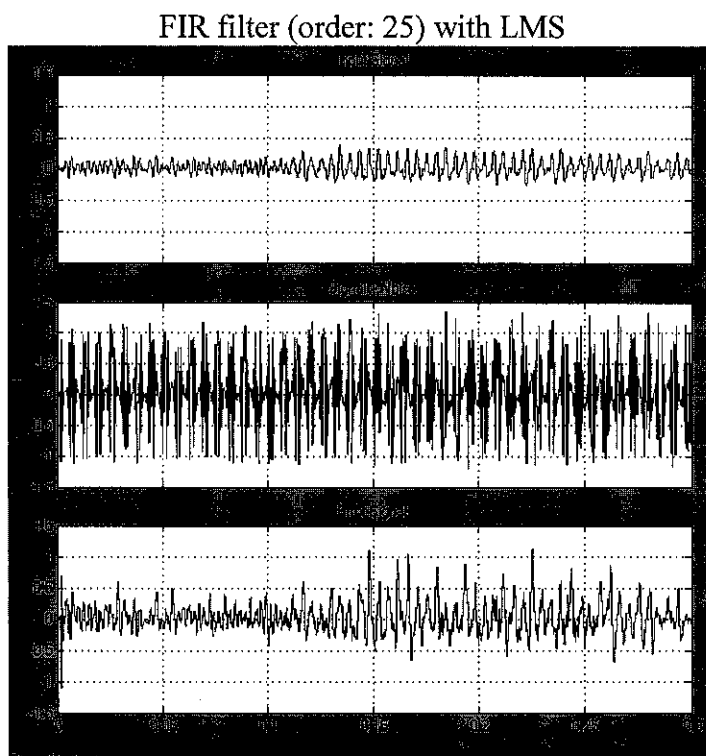


Figure G.4: Input signal, interfered signal and output signal of adaptive noise cancellation system with LMS algorithm and 25 orders FIR filter.

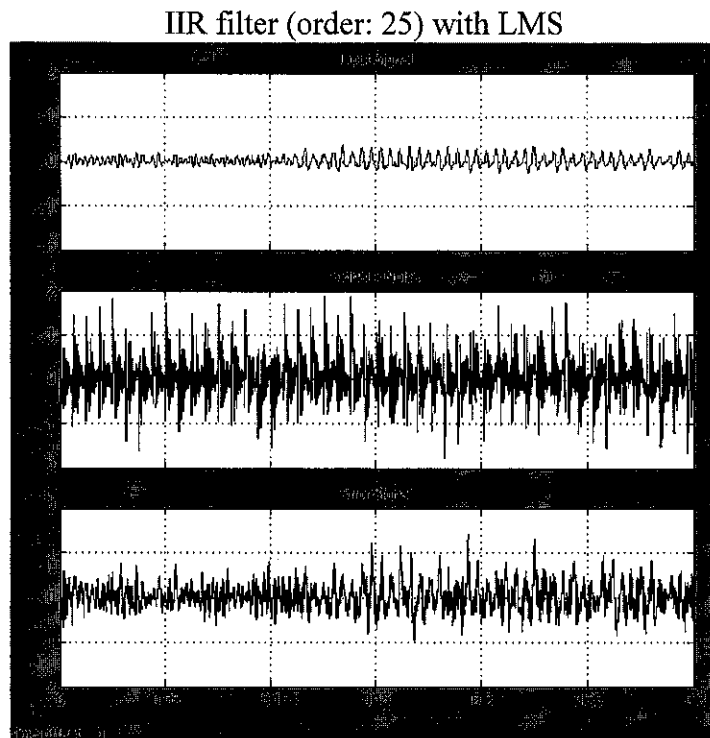


Figure G.5: Input signal, interfered signal and output signal of adaptive noise cancellation system with LMS algorithm and 25 orders IIR filter.

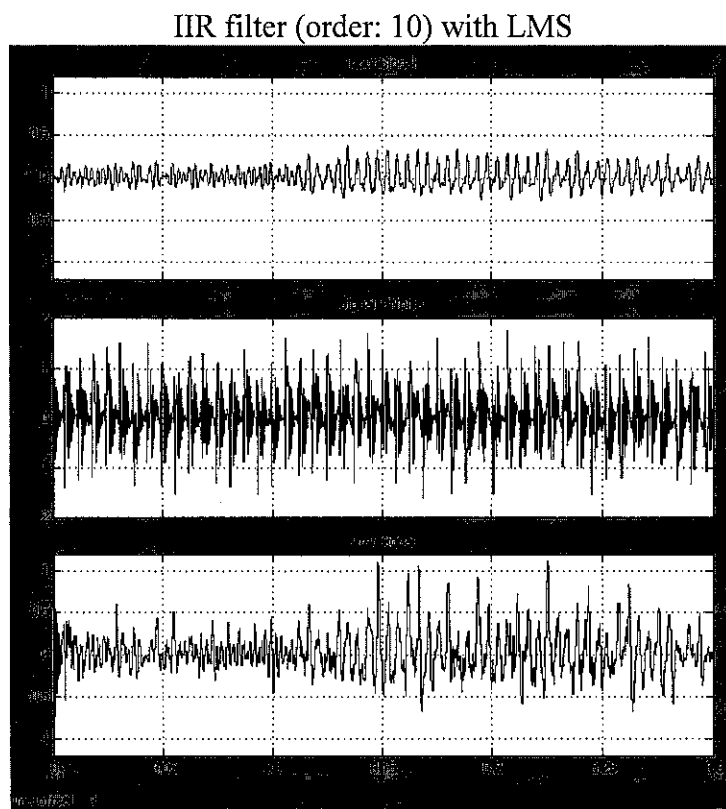


Figure G.6: Input signal, interfered signal and output signal of adaptive noise cancellation system with LMS algorithm and 10 orders IIR filter.

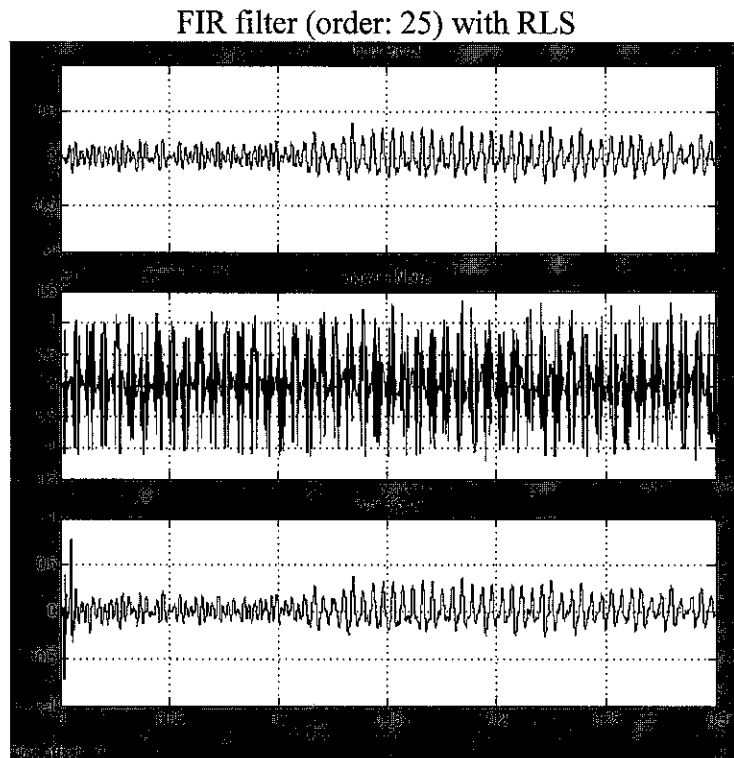


Figure G.7: Input signal, interfered signal and output signal of adaptive noise cancellation system with RLS algorithm and 25 orders FIR filter.

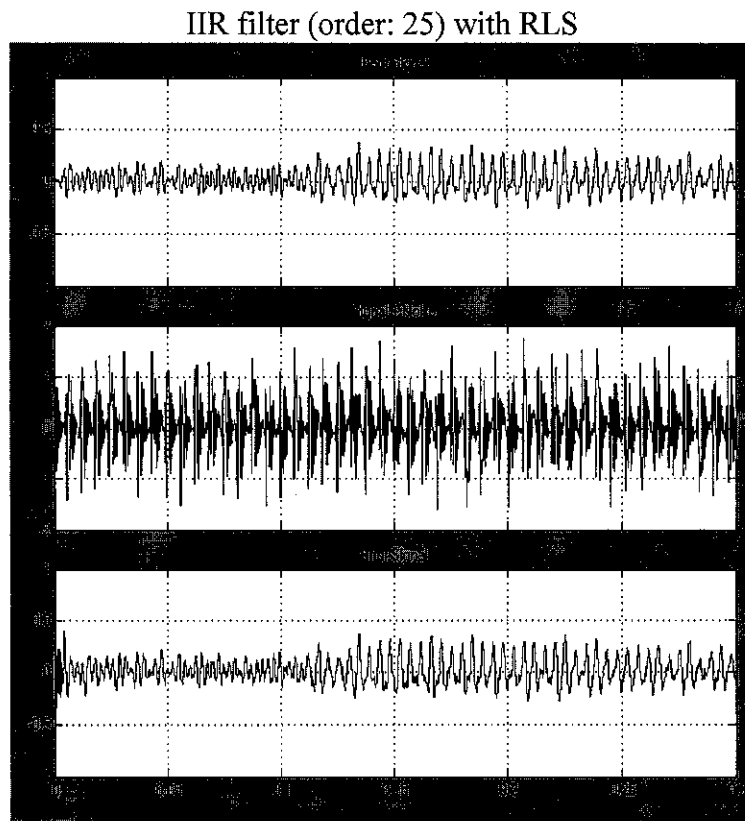


Figure G.8: Input signal, interfered signal and output signal of adaptive noise cancellation system with RLS algorithm and 25 orders IIR filter.

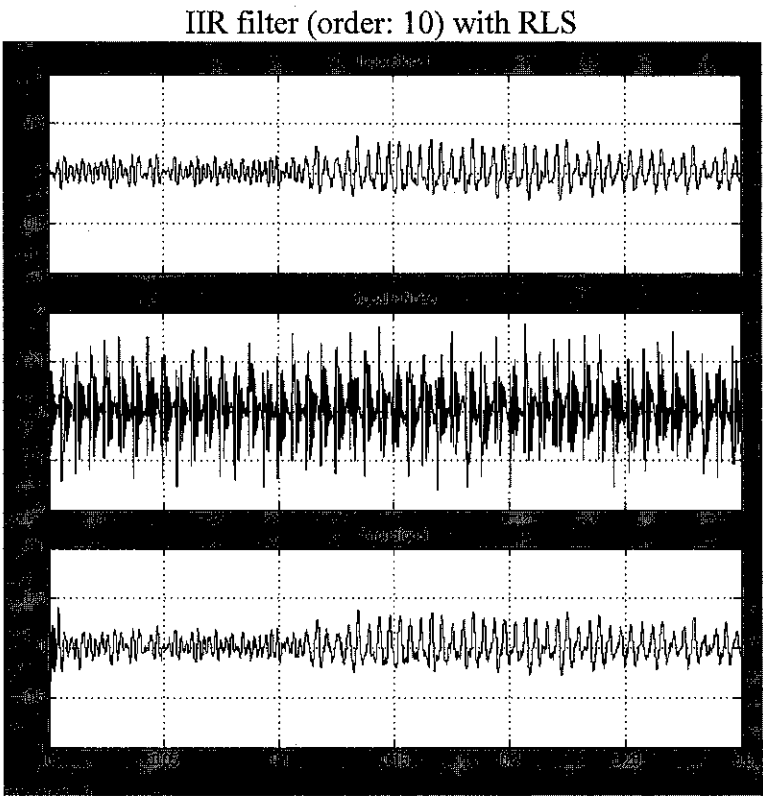


Figure G.9: Input signal, interfered signal and output signal of adaptive noise cancellation system with RLS algorithm and 10 orders IIR filter.